



Cat. No. 26-3821

# TRS-80<sup>®</sup>

---

## MODEL 100

### BASIC LANGUAGE LAB



CUSTOM MANUFACTURED IN U.S.A. FOR RADIO SHACK, A DIVISION OF TANDY CORPORATION

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE  
PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A  
RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

## LIMITED WARRANTY

### I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

### II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

### III. LIMITATION OF LIABILITY

- A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".  
NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.
- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

### IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on one computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on one computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of one computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may sell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

### V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

### VI. STATE LAW RIGHTS

The warranties granted herein give the original CUSTOMER specific legal rights, and the original CUSTOMER may have other rights which vary from state to state.

---

**TRS-80®**

---

**Model 100**

---

**BASIC Language Lab**

---

**Radio Shack®**

A DIVISION OF TANDY CORPORATION  
FORT WORTH, TEXAS 76102

Model 100 BASIC Language Lab Program:  
© 1983 Tandy Corporation  
All Rights Reserved.

Model 100 BASIC Language Lab Program Manual:  
© 1983 Tandy Corporation  
All Rights Reserved.

Reproduction or use, without express written permission from Tandy Corporation, of any portion of this manual is prohibited. While reasonable efforts have been taken in the preparation of this manual to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors or omissions in this manual, or from the use of the information contained herein.

10 9 8 7 6 5 4 3 2 1



# Contents

Introduction .....	1
Lesson #1 Introduction to BASIC .....	3
Lesson #2 Saving Programs .....	15
Lesson #3 Interest Calculations .....	31
Lesson #4 Sales Commissions .....	45
Lesson #5 Day, Time and Date .....	57
Lesson #6 Using the Editor .....	69
Lesson #7 Sales Trend .....	87
Lesson #8 Plot Your Data .....	99
Lesson #9 Functions .....	111
Lesson #10 Data Files .....	123
Lesson #11 Average Sales .....	131
Lesson #12 Sound & Simulation .....	145
Lesson #13 Function Keys .....	155
Lesson #14 Using the COM Option .....	165
Lesson #15 TELCOM Applications .....	175
Application #1 Calculator .....	185
Application #2 Memory Master Game .....	189
Application #3 Descriptive Statistics .....	195
Index .....	203

---



# Introduction

If you've used your TRS-80 Model 100 just once, you know how simple, versatile, and powerful a computer it is. Its built-in application programs allow you to perform normally complex computer operations with ease. This includes data manipulation, computer-to-computer communications, word processing, and more.

However, as you become more familiar with your Model 100, you can make the computer even more useful by customizing it to suit your own special needs. This is done through BASIC, the built-in programming language.

For instance, from BASIC, you can:

- Re-define the Function Keys (F1) through (F8).
- Communicate with information services and other computers.
- Write programs for a wide range of applications such as forecasting sales trends and performing interest or mortgage calculations.
- Make use of the computer's graphic and sound capabilities.

and a host of other operations!

This course will show you how to perform operations such as these by explaining in detail the BASIC section of your Model 100 Owner's Manual. This means that by the time you've finished this course, you'll be writing your own programs and using the built-in application programs more effectively.

Since most of the application programs not built-in will be written in BASIC, and since BASIC interacts with the other built-in programs, you'll find it is definitely to your advantage to become familiar with BASIC.

So sit back and get ready to enjoy your Model 100 even more. You're about to find out how powerful a computer it really is!



# Lesson #1 Introduction to BASIC

To use the BASIC capabilities of the Model 100, you must first learn how to communicate with your Computer. Essentially, this involves typing instructions on the keyboard and watching the display for responses from the Computer.

While you can type anything you wish on the keyboard, the Model 100 only responds to words written in its own "language." This "language" is **BASIC**. If you type something which the Model 100 does not recognize as a BASIC word, it will respond with an error message.

In this lesson, you will learn a few BASIC commands to communicate with your Computer and write simple programs.

It should be mentioned at this point, that even when you type a BASIC word, the Model 100 will not respond until the ENTER key (located in the right side of the keyboard) is pressed. After pressing **ENTER**, the line just typed is placed in memory for processing by the Computer.

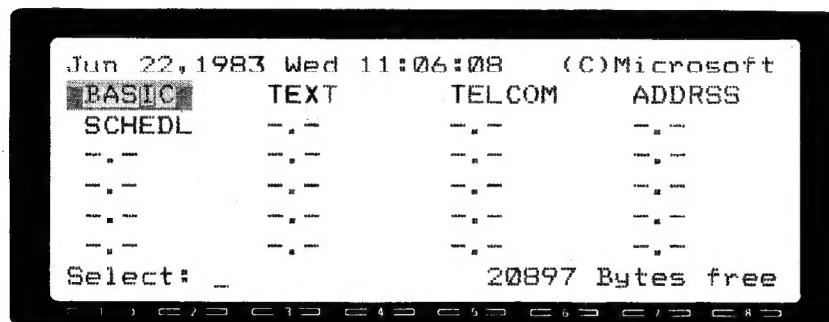
Throughout this Manual, the phrase "enter this command" will be used often. This simply means that you should press **ENTER** after typing the command or instruction. Also, when we tell you to press **BREAK**, you should press both **SHIFT** and **PAUSE** together.

## Accessing BASIC

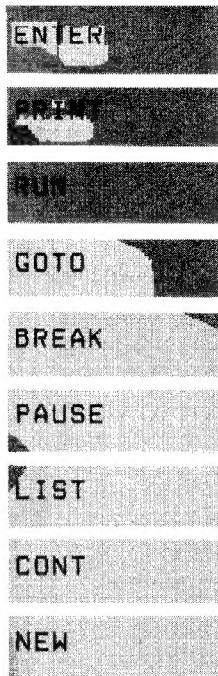
Prior to typing or editing a BASIC program, you must access the BASIC interpreter of the Model 100.

When you Power-Up the Computer, you will see the Main Menu, which shows all the "files" that exist in the Model 100's memory. Think of these computer files simply as file folders that may hold text or programs.

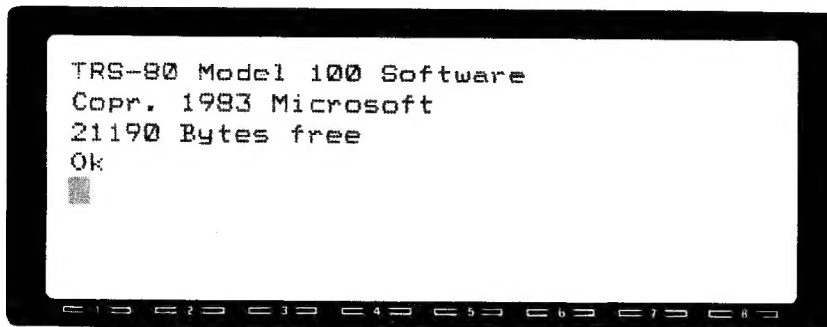
If you haven't created any files, the Main Menu displays the built-in application programs:



On initial Power-Up, the Main Menu Cursor — the large, dark rectangle — is positioned over the word "BASIC." The Cursor can be placed on any other Menu word by pressing the Cursor Movement Keys (←, →, ↑, or ↓).



To access the BASIC interpreter, simply position the Cursor over the word BASIC and press **(ENTER)**. The Display will then look like this:



The number 21190 indicates the number of free bytes for creating any programs and it may vary depending upon the capacity of your machine and whether any other files have been created and saved.

The word OK and the flashing cursor symbol below it, indicate that you are in the **Command Mode** of BASIC and ready to begin programming.

## Experiment #1 Entering a Command

A **command** is an instruction to the Computer ordering it to do something *immediately*. In this experiment you will learn how to enter a simple command.

First you will attempt to have the Model 100 print out a name on the display. Type the name:

JOHN SMITH

You may use any other name. Now, press **(ENTER)**. As soon as you do this, the Computer displays the message:

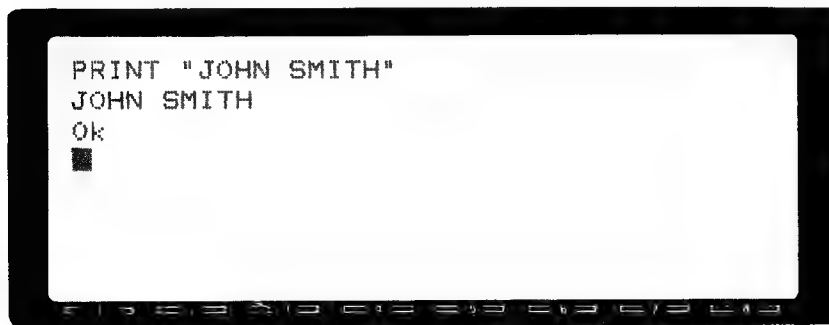
?SN Error

This message indicates that an error, specifically a syntax error, has been made. The syntax error occurred because the Computer doesn't recognize John Smith as part of its vocabulary (also known as the Instruction Set).

The correct way to instruct the Computer to print the name is to type:

PRINT "JOHN SMITH" **(ENTER)**

Be sure to press **(ENTER)** at the end of the line. The key word **PRINT** and the quotation marks enclosing the name are recognized as part of the Computer's vocabulary. This way no syntax error occurs and the name appears printed right below the command.



Use the PRINT command to print your own name if you haven't already done so. Also, try printing other phrases. For example, to print How now brown cow, type:

```
PRINT "HOW NOW BROWN COW" (ENTER)
```

Notice that the phrase is printed exactly as it appears within the quotation marks, including spaces.

## Experiment #2 What is a BASIC Program?

A **BASIC program** is a list of **instructions** (or statements) that the Computer executes, one at a time, in a *sequential order*. An instruction differs from a command in that it is preceded by a line number.

Here is a simple BASIC program:

```
10 PRINT "JOHN SMITH"
```

This program, consisting of only one statement, the PRINT statement, accomplishes the same thing as the command:

```
PRINT "JOHN SMITH" (ENTER)
```

But because the word PRINT is being used in a program and preceded by a line number, it is now called a statement. The number 10 which is typed before the PRINT statement is called a **line number**. *Every line in a BASIC program must have a line number, even if the program contains only a single line.*

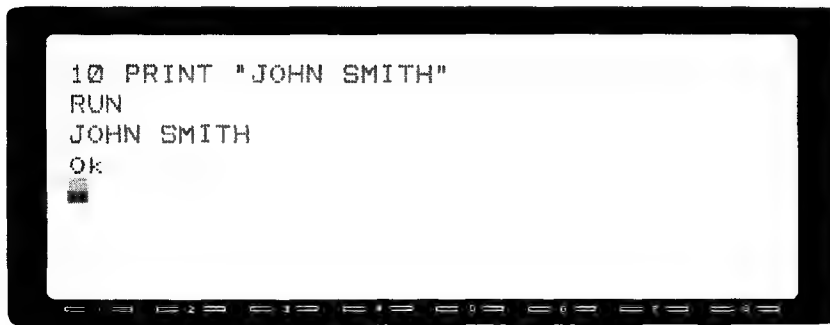
Type line 10 as it appears above. Notice that nothing happens this time when you press (ENTER). Unlike a command, which instructs the Computer to perform a task immediately, a program does nothing until the Computer is instructed to execute it. This is accomplished with the command:

```
RUN
```

Type the RUN command. Don't forget to press (ENTER) after typing it.

After entering the RUN command, the program is executed and the name is displayed.





A more economical way of instructing the Computer to execute your program is to press the **RUN Function Key, (F4)**. This accomplishes the same thing as typing RUN **(ENTER)**.

Now try a slightly more ambitious program. Enter the following two-line program:

```
10 PRINT "RADIO SHACK MODEL 100"  
20 GOTO 10
```

The second statement in this program begins with line number 20, indicating that this instruction should be executed after the first line which has a smaller line number. There is nothing special about the line numbers used in this program. The important thing is that the PRINT instruction has a smaller line number than the GOTO instruction.

Execute the program with the RUN command or by pressing **(F4)**.

As you can see, unlike the first single line program, this second program prints the name within quotation marks repeatedly. This is known as an **"infinite loop"** program because the GOTO statement in the second line of the program simply transfers control back to the first line which prints the **"string"** (a group of characters and/or numbers) again on the next line.

Because of this continuous transfer, the program has no way of terminating and so it must be terminated manually by you. To **"break"** the program, press **(BREAK)** **(SHIFT) (PAUSE)**.

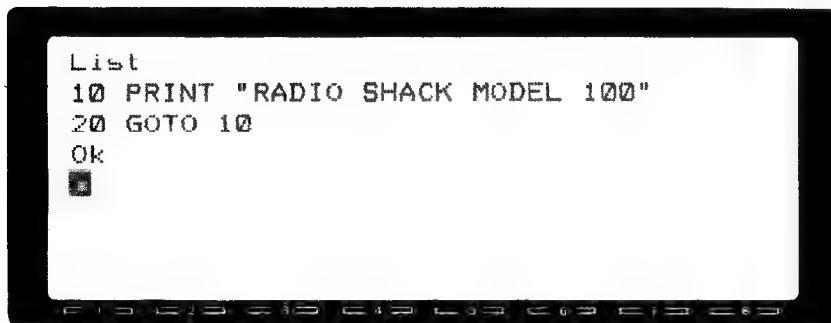
When you press **(BREAK)**, the Computer will display a message to show where the program stopped when it was **"broken."** For example:



Note that the program you wrote "scrolls up and off" the screen as the program began printing. If you wish to see the program as you wrote it again, enter the command:

**LIST**

or simply press the **LIST Function Key, (F5)**. In either case, the program will appear on the Display again.



Execute this program again with the **RUN** command or by pressing **(F4)**.

After letting the program run a few seconds, terminate it by pressing **(BREAK)**.

This time after pressing **(BREAK)**, enter the command:

**CONT**

As you can see, the program resumes execution. The **CONT** command is used to **CONT**inue execution after the program has been "broken." Execution will start at the same place where the program was interrupted.

## Experiment #3 Simple Editing

Here's the program from the previous experiment:

```
10 PRINT "RADIO SHACK MODEL 100"  
20 GOTO 10
```

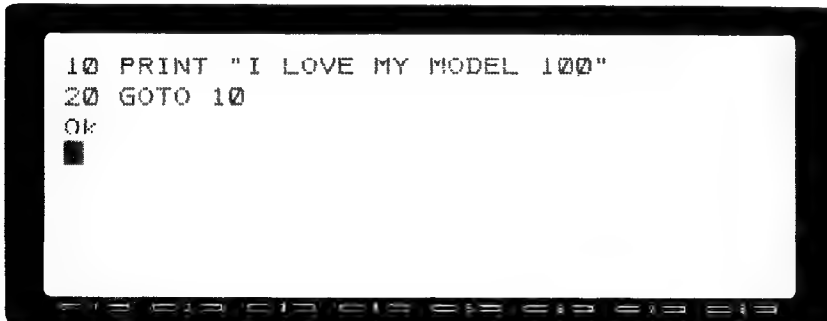
Suppose you want to change line 10 so that the message reads:

```
I LOVE MY MODEL 100
```

This can be done by retyping line 10 entirely:

```
10 PRINT "I LOVE MY MODEL 100" (ENTER)
```

Now, when you list the program with the LIST command or with the (F5) Function Key, the following is displayed:

A screenshot of a terminal window with a black border. Inside, the text is white. It shows a BASIC program listing: line 10 with 'PRINT "I LOVE MY MODEL 100"' and line 20 with 'GOTO 10'. Below the listing, the text 'Ok' is followed by a small black square cursor. At the bottom of the window, there is a row of small, light-gray icons representing various functions like back, forward, and search.

```
10 PRINT "I LOVE MY MODEL 100"  
20 GOTO 10  
Ok  
■
```

Execute this program to verify that the new phrase is displayed. Now, instead of pressing (BREAK) to interrupt execution, press the (PAUSE) key. This will cause the program to stop momentarily. To continue execution, simply press the (PAUSE) key again. Pausing a program may be useful when you want to read what is being displayed before it scrolls off view. Of course this program must still be terminated by pressing (BREAK) because it still is in an infinite loop.

Try a few other experiments with the PRINT command. Type in the following program:

```
10 PRINT "MY NAME IS"  
20 PRINT "LEE"
```

Run this program. The output should be:

A screenshot of a terminal window with a black border. Inside, the text is white. It shows the output of the previous program: 'MY NAME IS' on one line and 'LEE' on the next. Below the output, the text 'Ok' is followed by a small black square cursor. At the bottom of the window, there is a row of small, light-gray icons representing various functions like back, forward, and search.

```
MY NAME IS  
LEE  
Ok  
■
```

Now, retype line 10 to insert a space between the word "IS" and the quotation mark. Also, add a semicolon at the end of line 10. The changed program should look like this

```
10 PRINT "MY NAME IS ";  
20 PRINT "LEE"
```

Execute this program. The output should appear as:

```
MY NAME IS LEE
Ok
■
```

The printing appears all in one line because the semicolon instructs the Computer to continue printing immediately after the first line is printed. The space after "IS" in the first line was added so that the words "IS" and "LEE" would not run together.

If you wanted to space the name further apart, you could add more spaces after "IS" or you could add spaces before "LEE" in line 20.

Another way to space the printing is to use a comma instead of a semicolon. Retype line 10 so that it reads:

```
10 PRINT "MY NAME IS ",
```

Now list the program and it should read:

```
LIST
10 PRINT "MY NAME IS ",
20 PRINT "LEE"
Ok
■
```

Run this program. The output should appear as:

```
MY NAME IS      LEE
Ok
■
```

This time "IS" and "LEE" are spaced several columns apart. The comma in the first line means "begin printing in the next field" (more on fields later).

Enter the following program:

```
10 PRINT "HOW ", "NOW ";  
20 PRINT "BROWN"; "COW"
```

Would you guess what the output of this program will produce? When executed, the display will show:



The comma in line 10 caused the two strings HOW and NOW to be spaced several columns apart. The semicolon between the two words "BROWN" and "COW" caused them to print without a space. Now if line 10 is retyped so that the comma is changed to a semicolon and the semicolon at the end of the statement is omitted,

```
10 PRINT "HOW "; "NOW "  
20 PRINT "BROWN"; "COW"
```

the output would be



The second PRINT statement produces output on the second line because the carriage return after the first PRINT statement has not been suppressed with a semicolon or comma. If you wanted to print the words on one line, neatly spaced one column apart, you could rewrite your program as follows:

```
10 PRINT "HOW "; "NDW ";  
20 PRINT "BROWN "; "CDW"
```

When it is executed, it produces, as expected, the following:

```
HOW NOW BROWN COW
Ok
█
```

A BASIC program can be edited a line at a time simply by retyping the entire line as you have been doing.

To delete an entire line from a program, all you have to do is to type the line number of the statement you wish to delete and press **(ENTER)**.

For example, list the current program:

```
10 PRINT "HOW "; "NOW ";
20 PRINT "BROWN "; "COW"
Ok
█
```

Now type

20 **(ENTER)**

If you list the program again, you will see that line 20 has been effectively deleted:

```
10 PRINT "HOW "; "NOW ";
Ok
█
```

Retype line 20 to restore your program to its previous form:

```
10 PRINT "HOW "; "NOW ";
20 PRINT "BROWN "; "COW"
```

A new line can be added to a BASIC program at any time simply by typing it with the appropriate line number. If you want to add a statement before line 10, give it a number less than 10 (the smallest line number allowed is 1). If you want to add a line between the two lines, give it a number between 10 and 20 (e.g., 15).

Line numbers 10 and 20 were used to allow insertion of new lines. If successive line numbers had been used, for example 15 and 16, then no new lines could have been inserted. It is a good practice to use line numbers that are multiples of 10 (or at least 5).

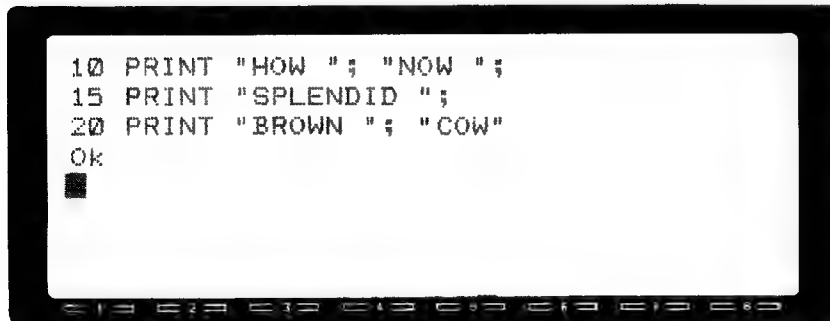
Here's your program again:

```
10 PRINT "HOW "; "NOW ";  
20 PRINT "BROWN "; "COW"
```

Suppose you want to insert a line between 10 and 20. Simply type:

```
15 PRINT "SPLENDID "; (ENTER)
```

When you list the program now, it will show:

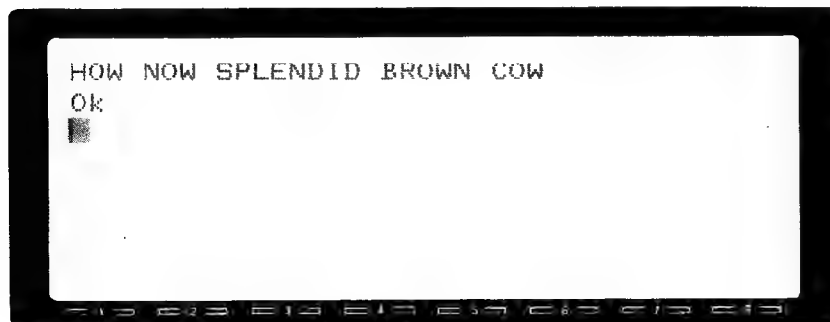
A screenshot of a BASIC program listing on a terminal. The text displayed is:

```
10 PRINT "HOW "; "NOW ";  
15 PRINT "SPLENDID ";  
20 PRINT "BROWN "; "COW"
```

Below the listing, the text "Ok" is visible, followed by a small black square cursor. The terminal window has a black border and a white background.

Even though line 15 was typed after lines 10 and 20, it takes its correct place in the program because its line number falls between 10 and 20.

Execute this program. The following output will result:

A screenshot of the output of the BASIC program on a terminal. The text displayed is:

```
HOW NOW SPLENDID BROWN COW
```

Below the output, the text "Ok" is visible, followed by a small black square cursor. The terminal window has a black border and a white background.

It should be clear by now that any BASIC program can be edited easily with the use of the line numbers. You can add, delete, insert, and change lines and that is all you ever need to do.



---

## Experiment #4 Writing Your Own Programs

By now you should be able to write simple BASIC programs using the two instructions PRINT and GOTO.

Before you go on experimenting with the spacing in PRINT statements using the comma and the semicolon, you should be aware of another useful command, the NEW command.

When you enter the NEW command, any program that has been previously typed and is currently residing in working memory will be erased automatically.

Before you begin typing in a new program, you should always use the NEW command to clear out the old program. Otherwise, you may end up with a combination of your new and old programs.

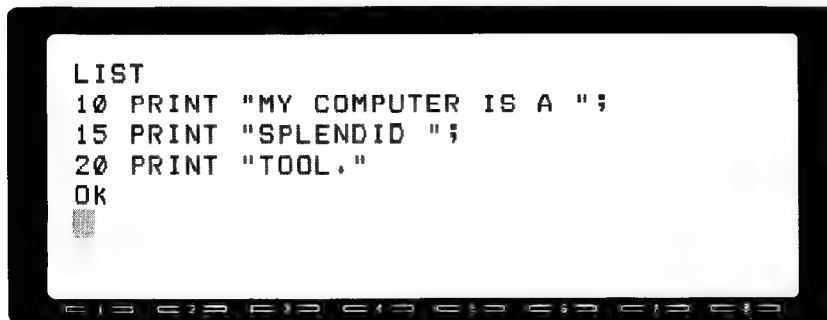
Assume that your old program still resides in memory:

```
10 PRINT "HOW "; "NOW ";  
15 PRINT "SPLENDID ";  
20 PRINT "BROWN "; "COW"
```

Now, without deleting this program, enter the following new program:

```
10 PRINT "MY COMPUTER IS A ";  
20 PRINT "TOOL."
```

If you list the program, you will find it is:



```
LIST  
10 PRINT "MY COMPUTER IS A ";  
15 PRINT "SPLENDID ";  
20 PRINT "TOOL."  
OK
```

Note that line 15 still exists because that line number was not used in the new program. So remember, before typing a new program, clear the memory with the command NEW. This won't be necessary, however, if you are certain that no program exists in memory.

### What you have learned:

In this lesson you have learned some commands to write and execute a simple BASIC program. The PRINT and GOTO statements have been used to display simple messages. Editing a BASIC program can be accomplished by retyping existing lines or typing new lines. The NEW command is used to delete an entire program from memory.

---



---

## Lesson #2 Saving Programs

In this lesson you will learn how to save programs in memory and on cassette tapes. You will also learn how to recall a program from storage and how to merge a stored program with another program.

### Experiment #1 Saving a Program In RAM

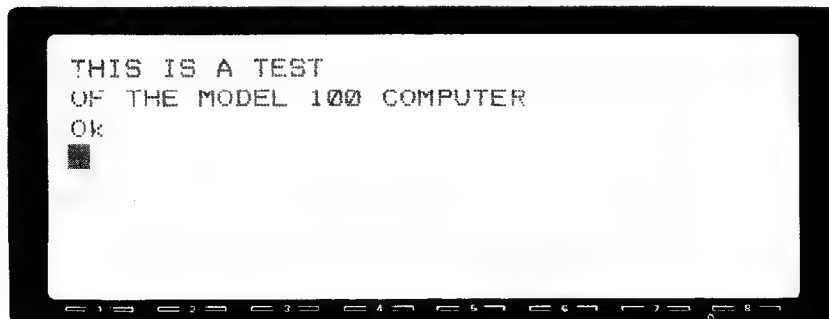
As it was mentioned before, the Model 100 can hold several files, many of which can be program files. In Lesson 1 you learned to write simple BASIC programs in the BASIC system of the Computer. When you have written a program that you intend to use repeatedly, it is a good idea to save it in memory (also known as RAM, for Random Access Memory). When you save a program in RAM, in effect, you create a program file.

The following program serves to demonstrate how any program can be saved in RAM:

```
10 PRINT "THIS IS A TEST"  
20 PRINT "OF THE MODEL 100 COMPUTER"
```

Access BASIC. Clear working memory with the NEW command and then type the program as it appears above.

Execute the program. The following will be displayed:



The first step for saving a program in RAM is to decide upon a *filename*. This filename appears listed as a file in the Main Menu and it serves to identify the program. Filenames cannot exceed six characters in length.

You may use any combination of letters and digits for a filename. However, the first character in a filename must always be a letter. The following are examples of valid filenames:

```
MYPROG  
SKETCH  
ACCNT2
```

SAVE

LOAD

FILES

MERGE

KILL

NAME

CSAVE

CLOAD

CLOAD?

The following are examples of illegal filenames:

1PROG (must begin with a letter)  
MICROCOMPUTER (exceeds the maximum of six characters in any file name)

Suppose you want to save the program above under the filename

PROG1

To do this, simply type:

SAVE "PROG1" (ENTER)

The Display will show the prompt OK to indicate that the program has been SAVED.

Use the command NEW to clear the program from working memory. To verify that the program no longer exists in working memory, enter the command:

LIST (ENTER)

or press (F5), the LIST Function Key. The computer will respond with:

LIST  
OK

and nothing else, indicating that there is no program currently in working memory.

The program has not been wiped out. It has been erased from working memory but it now exists in RAM. To confirm that this is true, enter the command:

FILES (ENTER)

or press the FILES Function Key, (F1). In response to this command, the names of all the files stored in RAM, including all BASIC programs, will be displayed. In this case, if you haven't SAVED any other programs or files, the name

PROG1.BA

will be displayed.

This is your program. The characters ".BA" form a "file extension" which indicates that this file is a BASIC program. The Computer automatically appends this extension to the name of any BASIC program when it is saved in RAM.

If you want more proof that your program was indeed saved as a file, press the Menu Function Key, (F8). You will see PROG1.BA displayed in the Main Menu as shown below.



---

A simpler way to save a program in RAM, is to use the SAVE Function Key, (F3). After typing a program you wish to save, simply press (F3). The Computer will prompt you with the message:

Save "

All you have to do then is to type a name for the file as you did before and press (ENTER).

The number of files which can be saved is limited only by the amount of RAM available. If you continue to add files to RAM, eventually all available RAM will be used up and no more programs can be saved.

## Experiment #2 Loading a Program From RAM

After you have saved a program in RAM, you may execute it simply by positioning the Cursor over the word identifying it in the Main Menu and pressing (ENTER).

However, if you wish to modify or alter the program in any way, it is very convenient to LOAD it into the BASIC system.

Let's use PROG1 which you SAVED in the last experiment and LOAD it into the BASIC system.

You can do that with the command:

LOAD "PROG1" (ENTER)

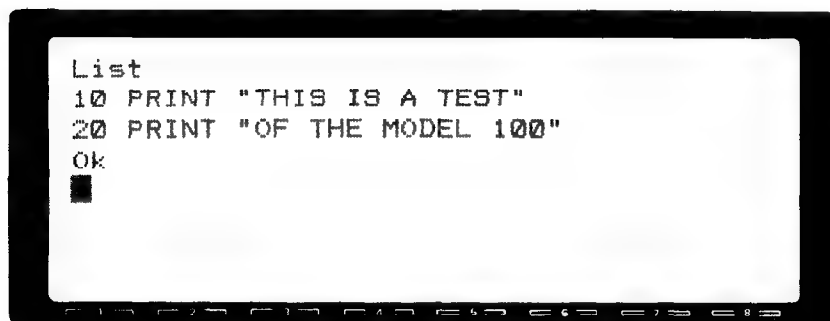
Another way to LOAD the program is to press the LOAD Function Key, (F2). After pressing (F2) the message

Load "

appears on the Display. All you should do then is to type

PROG1" (ENTER)

After Loading PROG1 with the LOAD command or (F2), LIST the program to verify that it is indeed in working memory. The following should be displayed:



```
List
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100"
Ok
■
```

If you execute the program with the RUN command you will see that the output is the same as before.

Enter the command FILES or press (F1).

---

You will see that the program PROG1.BA is still listed, indicating that it is still saved in RAM. Loading a program from RAM does not erase it from storage.

Notice, however, that an asterisk appears to the right of PROG1.BA. The asterisk indicates that the program is currently in working memory.

Use the NEW command to delete the program PROG1 from working memory. Use the LIST command to verify that working memory does not contain the program. Now type the command

```
RUN "PROG1" (ENTER)
```

The following should appear on the LCD:

A rectangular LCD screen with a black border. The screen displays the text "THIS IS A TEST" on the first line, "OF THE MODEL 100 COMPUTER" on the second line, and "Ok" on the third line. Below "Ok" is a small black square cursor. At the bottom of the screen, there is a row of small, faint icons or characters.

```
THIS IS A TEST  
OF THE MODEL 100 COMPUTER  
Ok  
█
```

This illustrates a useful option of the RUN command. If the RUN command is followed by the name of a program stored in RAM (the name must be enclosed in quotes), then the program will be loaded into working memory and executed immediately. Thus the command:

```
RUN "PROG1" (ENTER)
```

is equivalent to the two commands:

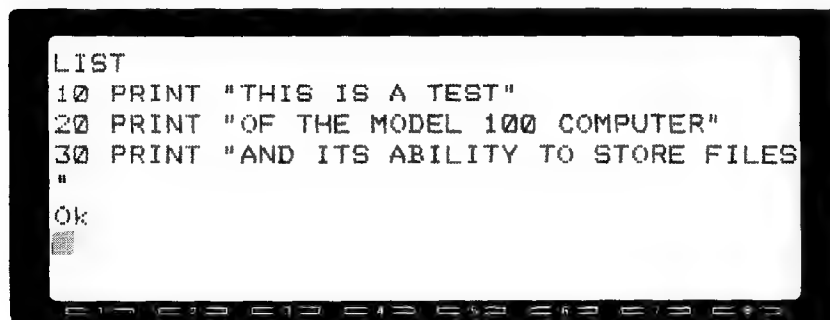
```
LOAD "PROG1" (ENTER)  
RUN (ENTER)
```

Use the LIST command to verify that the program now resides in working memory.

The program will be modified by adding a third line. Type the following line:

```
30 PRINT "AND ITS ABILITY TO STORE FILES" (ENTER)
```

List the program to verify that it is:

A rectangular LCD screen with a black border. The screen displays the text "LIST" on the first line, "10 PRINT \"THIS IS A TEST\"" on the second line, "20 PRINT \"OF THE MODEL 100 COMPUTER\"" on the third line, "30 PRINT \"AND ITS ABILITY TO STORE FILES\"" on the fourth line, and a quote character on the fifth line. Below the quote character is "Ok" on the sixth line, and a small black square cursor on the seventh line. At the bottom of the screen, there is a row of small, faint icons or characters.

```
LIST  
10 PRINT "THIS IS A TEST"  
20 PRINT "OF THE MODEL 100 COMPUTER"  
30 PRINT "AND ITS ABILITY TO STORE FILES"  
"  
Ok  
█
```

When you LOAD a program from RAM into working memory, you may add, delete, or insert new lines as you wish. The changes you make are immediately incorporated into the program.

Delete the program from working memory with the NEW command. Use the LOAD command to recall program "PROG1" from RAM. Type:

LOAD "PROG1" (ENTER)

List the program with the LIST command. You should see:

```
LIST
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
"
OK
```

The new line was effectively incorporated in "PROG1" which is stored in RAM.

You can also LOAD and execute a BASIC program directly from the Main Menu.

Press (F8) to exit BASIC and return to the Main Menu. You should see something similar to

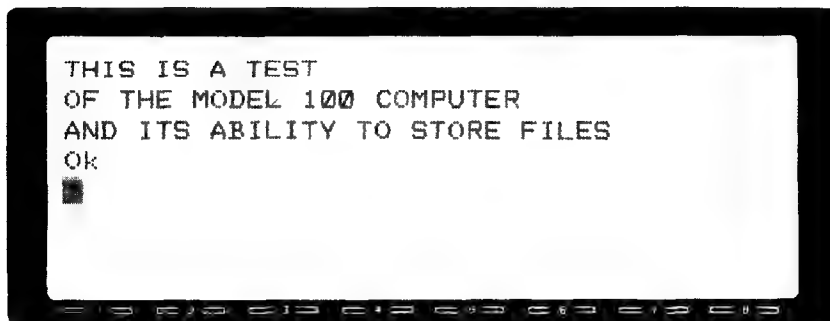
```
Jun 22, 1983 Wed 10:37:07 (C)Microsoft
TEXT TELCOM ADDR55
SCHEDI PROG1.BA -- --
-- -- -- --
-- -- -- --
-- -- -- --
-- -- -- --
Select: _ 21350 Bytes free
```

Move the cursor over the filename PROG1.BA as indicated below.

```
Jun 22, 1983 Wed 14:52:17 (C)Microsoft
BASIC TEXT TELCOM ADDR55
SCHEDI PROG1.BA -- --
-- -- -- --
-- -- -- --
-- -- -- --
-- -- -- --
Select: 21358 Bytes free
```



If you press **(ENTER)**, several things will happen: the computer enters BASIC and then loads and executes PROG1.BA. You will see



with the Ok prompt indicating that you are in BASIC. If you list the program you will see that PROG1.BA is in working memory.

## Experiment #3 Changing a Filename

It is possible to change a file name using the command:

```
NAME "old filename.extension" AS "new filename.extension"
```

where *old filename* is the name of the program as it now exists, and *new filename* is the new program name you wish to assign to it.

For example, if you wish to change the filename "PROG1" to "TEST1," type:

```
NAME "PROG1.BA" AS "TEST1.BA" (ENTER)
```

Verify the name change with the FILES command or by pressing **(F1)**.

Now, using the NAME. . . AS command, change the file name back to "PROG1."

```
NAME "TEST1.BA" AS "PROG1.BA" (ENTER)
```

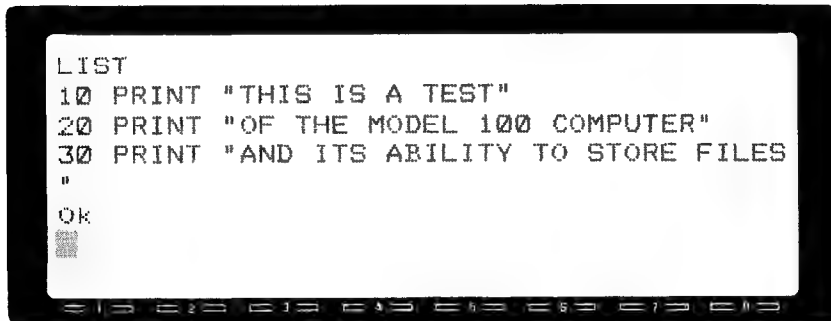
Again, you may confirm that the name was changed to PROG1 with the FILES command.

## Experiment #4 Saving a Program on Cassette

While it is convenient to save your programs in RAM, there is only a finite amount of space available. Eventually, there will be no room left for new programs to be saved.

An alternate method of saving your files is to store them on cassette tapes using a cassette recorder. Using cassettes, you can store essentially an unlimited number of programs.

PROG1 should still be in working memory. Use the LIST command to confirm that it is. It should appear as:



```
LIST
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
"
OK
```

If it is not in memory you should LOAD it from RAM or type it in as shown above.

To save a program on cassette, it is first necessary to connect the Model 100 to a suitable cassette recorder. For optimum results we recommend the Radio Shack CCR-81 Computer Recorder (catalog number 26-1208) with connecting cable and instructions supplied. Be sure that the proper connections are made before proceeding further.

Place a blank tape in the cassette recorder and rewind it, if necessary. Then advance the tape past the leader. (If you use Radio Shack Leaderless cassettes, catalog number 26-3019, this isn't necessary.)

Press the RECORD (red key) and PLAY keys down together. They should stay down, but the tape will not move. If it does, you do not have the remote jack inserted. Insert the jack.

Now you are ready to save the program onto the tape. Enter the following command:

```
CSAVE "PROG1"
```

The recorder will run briefly and then stop. The RECORD and PLAY keys, however, will stay down. After the tape stops, you may press the STOP key and rewind the tape.

The program now has been saved with the filename "PROG1."

Another way to save the program on tape, instead of using CSAVE "PROG1" is to use the command:

```
SAVE "CAS:filename"
```

where CAS: specifies the device to be used for the saving operation, in this case the cassette recorder, and filename is the name of the program to be saved. Using this command, you would specify:

```
SAVE "CAS:PROG1" (ENTER)
```

## Experiment #5 Loading a Program from Cassette

The program you just saved on tape in the previous experiment, PROG1, will now be loaded back into working memory. But first, delete PROG1 from working memory with the NEW command.

Verify that it has been deleted with the LIST command or by pressing the LIST Function Key, (F5).

Be sure that the cassette recorder is properly attached to the Computer. Insert the cassette containing the program into the recorder and rewind if necessary.

Press PLAY on the recorder. The key will stay down, but the tape will not advance.

Enter the following command:

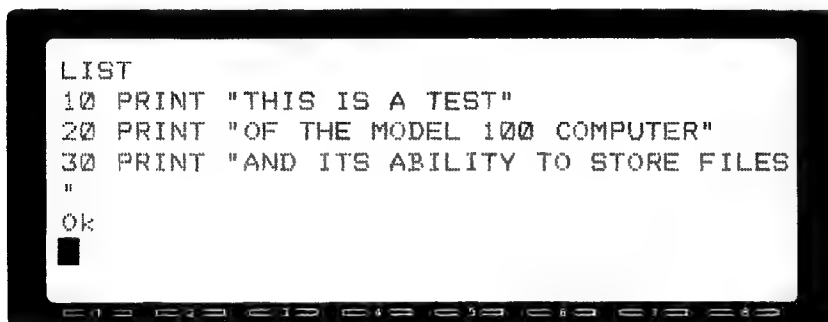
```
CLOAD "PROG1"
```

As soon as you press (ENTER), the tape will start turning and the program will be read into the Computer. If this is accomplished successfully, the computer will display:



If the tape was not read successfully, an I/O (Input/Output) error message will be displayed. If this happens, rewind the tape and adjust the volume control. Then use the CLOAD command again to read the tape.

If the tape is read successfully, you may verify that the program is in working memory by listing it with the LIST command (or with (F5)). The following will be displayed:



---

When the command CLOAD "PROG1" is entered, the Computer searches the tape until a program stored under the name "PROG1" is found. This program is then read into working memory.

The command

LOAD "CAS:filename" (ENTER)

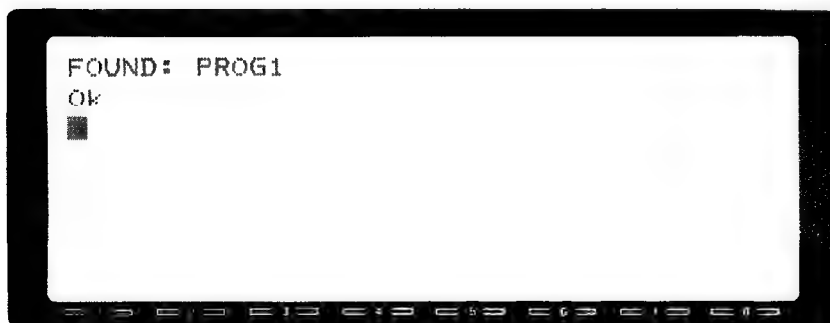
may be used in place of CLOAD "filename". Remember that filename specifies the name of the program you wish to LOAD from the cassette.

Delete the program from working memory with the NEW command and verify that it has been erased by using the LIST command.

Rewind the cassette and press the play key. This time type the command:

CLOAD (ENTER)

When this command is entered, the cassette recorder will run briefly and then stop. Also, the computer will display:



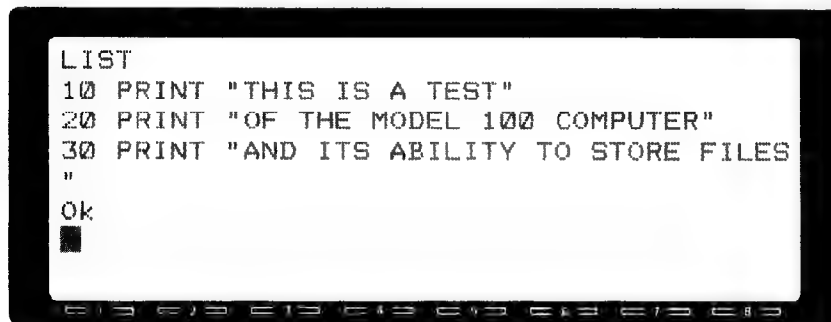
indicating that the program was read successfully. List the program in memory with the LIST command. You will see that the same program was loaded from the tape.

This illustrates another option of the CLOAD command. When the filename of the program is omitted, the first program encountered on the tape is loaded into memory. Since your program PROG1 was the first program on the tape, it was loaded into the Computer.

## Experiment #6 Verifying a Stored Program

Your Model 100 allows you to verify that a program has been saved successfully (i.e. without any errors) on cassette tape. Another option of the CLOAD command is used to accomplish this.

Verify that the program still resides in working memory with the LIST command. If it has been deleted, load it from the cassette tape or type it in from the keyboard. Here is the listing again:



```
LIST
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
"
Ok
```

Rewind the cassette tape containing the program and press the PLAY key on the recorder.

Type the command

CLOAD? "PROG1" (ENTER)

The recorder will run briefly and then stop. The Computer will display:

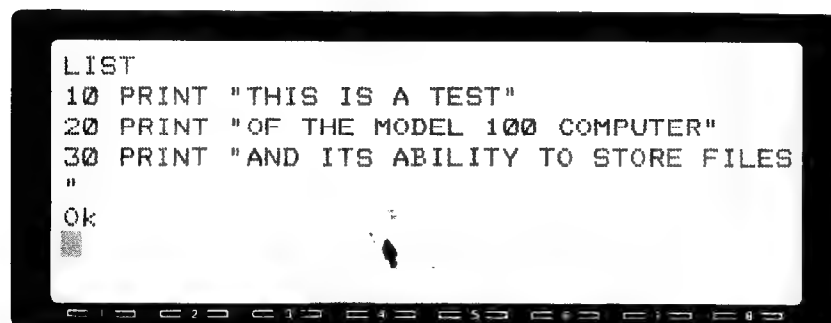
```
Found: PROG1
OK
```

In this case, the program stored on tape was not SAVED into working memory, but was compared, character by character, with the program already in working memory. If an inconsistency is found at any point, an error message will be displayed, indicating that the program saved on the cassette tape was not the same as the one in working memory.

If an error has occurred, you can reSAVE the program and use the CLOAD? command again to check the saved program.

As with the CLOAD command, if the file name is omitted from the CLOAD? command, the first program encountered on the tape is compared with the program in working memory.

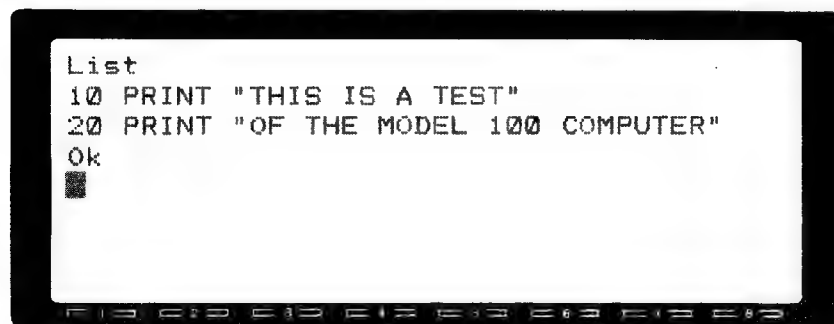
List the program in working memory. You should see:



```
LIST
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
"
Ok
```

---

Delete line 30 by entering the line number. Now list the program to verify that it is:



```
List
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
0k
```

This is not the program that was saved on the cassette (line 30 is missing). The CLOAD? command will be used to compare the program in working memory to the program saved on cassette. Since they are not the same, an error message will be displayed.

Rewind the cassette and press the play key. Enter the command

```
CLOAD?
```

This time, the message:

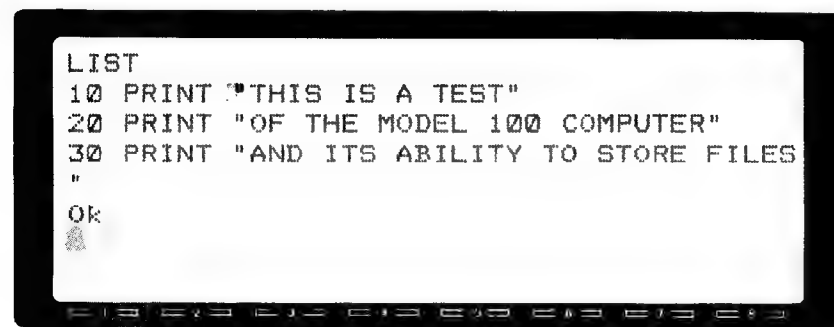
```
Verify failed
```

will be displayed, indicating that the two programs are not the same.

Don't forget the question mark (?) after CLOAD since otherwise the program saved on tape will be loaded into working memory, replacing the resident program there. If the CLOAD? command indicates a "bad" program on tape, then the CSAVE command can be used again to resave it.

## Experiment #7 Merging two Programs

Rewind the cassette tape containing the saved program and press the PLAY key. Then load the program back into memory with the CLOAD command. List the program to confirm that it is:



```
LIST
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
"
0k
```

The program will now be saved in RAM in "ASCII" format, rather than the usual "compressed" format. This means that the program will be saved exactly as it was written. Ordinarily, programs are saved in compressed form. In compressed form, key words such as PRINT, are replaced by a single character to conserve storage space.

Type the following command:

```
SAVE "PROG2",A (ENTER)
```

The "A" following the name indicates that the program is to be saved in ASCII mode.

Another way of saving a program in ASCII format, is to use the extension .DO as part of the SAVE command. For example, to SAVE PROG2 in this way you could type:

```
SAVE "PROG2.DO" (ENTER)
```

Delete the program from working memory with the NEW command.

Confirm that it has been saved in RAM with the FILES command.

PROG2 will be listed as:

```
PROG2.DO
```

The extension "DO" (for DOcument) will be attached to any BASIC program saved with the ASCII option. There are several reasons why you might want to save a file in the ASCII format. As it will be illustrated later in this experiment, merging files is one of the main ones.

Delete the program from working memory with the NEW command, and type the following one line program.

```
5 PRINT "HELLO MODEL 100 USER" (ENTER)
```

Merge PROG2, which is saved in RAM with the ASCII option, with this one line program, by typing the command:

```
MERGE "PROG2" (ENTER)
```

List the program in working memory to confirm that it is

```
5 PRINT "HELLO MODEL 100 USER"  
10 PRINT "THIS IS A TEST"  
20 PRINT "OF THE MODEL 100 COMPUTER"  
30 PRINT "AND ITS ABILITY TO STORE FILES"
```

Merging a program in working memory with one that has been saved in RAM, can only be done if the program in RAM was saved with the A (ASCII) option.

Delete lines 20 and 30 by entering just the line numbers and then rewrite line 10 as follows:

```
10 PRINT "HERE IS A MESSAGE FOR YOU" (ENTER)
```



List the program in working memory to confirm that it is:



```
LIST
5 PRINT "HELLO MODEL 100 USER"
10 PRINT "HERE IS A MESSAGE FOR YOU"
OK
```

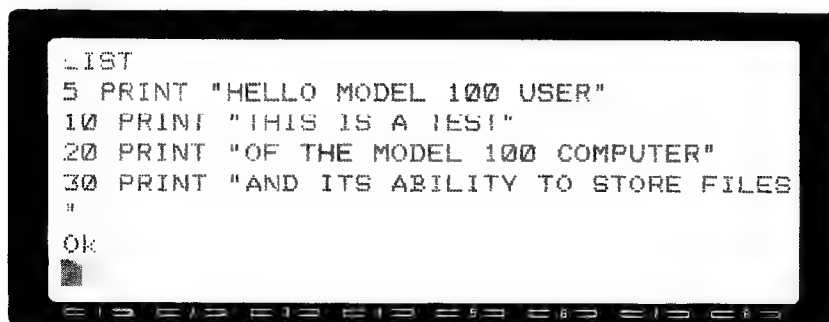
PROG2 will now be merged with this two line program. Remember that PROG2 is in RAM and consists of:

```
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
```

Note that the program in working memory and PROG2 both have a line 10. Merge the two programs with the command

```
MERGE "PROG2" ENTER
```

List the program. It will be:



```
LIST
5 PRINT "HELLO MODEL 100 USER"
10 PRINT "THIS IS A TEST"
20 PRINT "OF THE MODEL 100 COMPUTER"
30 PRINT "AND ITS ABILITY TO STORE FILES"
OK
```

Line 10 of the program (PROG2) saved in RAM replaced line 10 of the program resident in working memory. This will always happen. If two programs to be MERGED have any line numbers in common, the lines of the program SAVED in RAM will replace the lines of the program in working memory.

Programs saved on cassette tape may also be merged with a program in working memory as long as they are saved with the A option. The appropriate command is:

```
CSAVE "PROG2",A
```

or

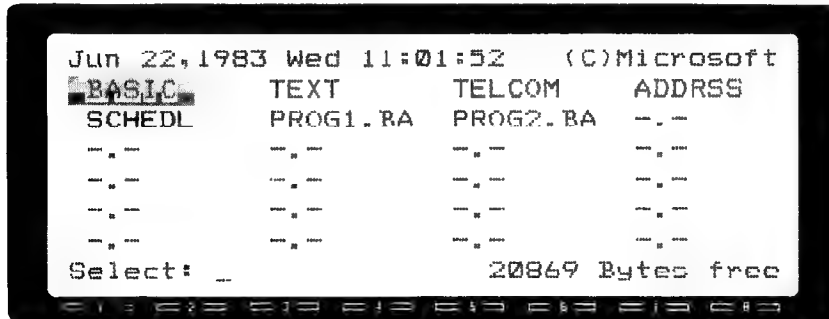
```
SAVE "CAS:PROG2",A
```

Then, to merge a program in working memory with one stored on tape, use the command:

```
MERGE "CAS:PROG2" (ENTER)
```

Once you have MERGE<sup>d</sup> PROG2 with the program in working memory, SAVE the newly formed program in the conventional way (without the A option) using the SAVE "PROG1" command.

If you return to the Main Menu by pressing (F8), you will see the two programs that have been SAVE<sup>d</sup>:



## Experiment #8 Deleting a file from RAM

Often times you'll need to eliminate files and programs that are no longer useful, or that you have SAVE<sup>d</sup> on tape to free additional memory space.

You can delete a file from the RAM storage area with the KILL command. The general form of the KILL command is :

```
KILL "filename.extension"
```

where *filename* is the name of the program you wish to delete, and *extension* specifies the characters .BA, in the case of a BASIC program, .DO, in the case of a program SAVE<sup>d</sup> with the A option or a text document, and .CO, in the case of a machine language program.

Use the KILL command to delete PROG1 from RAM:

```
KILL "PROG1.BA" (ENTER)
```

The prompt Ok and the blinking cursor will appear on the Display after the program has been deleted.

Use the FILES command (or (F1)) to list the files in RAM storage. You will notice that PROG1.BA is no longer listed. This is because the program has been deleted from RAM.

To summarize, the KILL command deletes a file from RAM storage. The NEW command deletes a program from working memory, but does not affect the file in RAM storage.

---

Also, you should be aware that a file cannot be deleted if the same file is also in working memory. That is, if you LOAD a program from RAM into working memory and then attempt to KILL this file, you will get an error message. You should first delete the program in working memory with the NEW command and then proceed to KILL the file in RAM.

## **What you have learned:**

You have learned how to SAVE BASIC programs in RAM and on cassette tape. You learned how to LOAD a program from RAM or from tape and how to verify that a program has been saved correctly. You also learned how to MERGE a SAVED program with a program in working memory. You found out that in MERGE operations, if line numbers in RAM are in common with line numbers in working memory, the line numbers of the RAM file take precedence over those in working memory. Finally, you learned how to delete files from working memory and from RAM.



## Lesson #3 Interest Calculations

This lesson illustrates the use of variables and the assignment of values to variables. Applications to the calculation of interest and mortgage payments will serve as useful examples of the use of variables in BASIC programs.

The programs used in this and all the following lessons will be explained line by line to give you a thorough understanding of the operations and the concepts involved.

### Experiment #1 Simple interest

If  $P$  dollars are borrowed at the simple interest rate  $r$ , then the amount  $S$  that must be repaid after  $t$  years is given by the formula

$$S = P(1 + rt)$$

$S$  is called the sum and  $P$  is the principal. The first program in this lesson calculates  $S$  for the following values of  $P$ ,  $r$  and  $t$ :

$P = \$10,000$   
 $r = 18\%$  per year  
 $t = 10$  years

Carefully enter the following program into the Computer:

```
10 P = 10000
20 R = .18
30 T = 10
40 S = P * (1 + R * T)
50 PRINT "THE SUM IS $"; S
```

After the program is entered, execute it by entering the RUN command or pressing the RUN Function Key, (F4). The output from the program should be:

```
THE SUM IS $ 28000
Ok
```

If \$10,000 is borrowed for 10 years at the simple interest of 18%, you will have to pay back \$28,000.

## How the simple interest program works:

**Line 10** assigns the value of 10000 to the variable P. This is called an *assignment statement*. The letter P is called a *numeric variable* because numerical values can be assigned to it. Note that the principal, \$10,000, is written without the dollar sign (\$) or comma (.). In general, numeric constants should be written without commas or dollar signs in BASIC.

**Line 20** assigns the value of .18 to the variable R. This is the decimal equivalent of the 18% interest rate. The conversion to decimal form is necessary because BASIC does not allow numeric constants to be written with a percent (%) symbol.

**Line 30** assigns the value of 10 to the variable T. This is the length of time (10 years) allowed to repay the loan.

**Line 40** computes the sum of principal and interest and assigns it to the variable S. Note that the expression,

$$P * (1 + R * T),$$

looks very much like the formula previously seen for calculating the sum. The symbol "\*" is used to denote multiplication and, as usual, + denotes addition.

**Line 50** prints an explanatory message and the value of S.

The variables P, R, T and S are all valid examples of numeric variables. A numeric variable can be

- a) any letter
- b) any two letters
- or
- c) a letter followed by a number.

For example, the following are all valid numeric variables:

A    AA    B2    CQ    Y9

But the following are not valid numeric variables:

1S    \$5    A?

The first four lines of the program are assignment statements. In general, an assignment statement follows the form:

$$\text{variable} = \text{expression}$$

where "variable" is any valid numeric variable and "expression" is any valid numeric expression. The right hand side can be a constant. The value of the expression is assigned to the variable.

The arithmetic operators in BASIC are:

+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation

Line 40 contains an example of a valid expression containing arithmetic operators. Here are some more:

- a)  $A * B / C$
- b)  $Q + A4 - 6.5$
- c)  $F * H8 + J9$

You should note that spaces or blanks are not significant in arithmetical expressions of this type. For example, the expression in a) could have been written

$$A * B / C \quad \text{or} \quad A * B / C \quad \text{or} \quad A * B / C$$

This allows you to space out the symbols so that they can be easily read.

However, it is important to note the order in which expressions such as c) are evaluated.

In BASIC, mathematical operations always follow a hierarchical order. Exponentiation has the highest priority, multiplication and division are next, and finally, addition and subtraction have the lowest.

Therefore, expression c) above, is interpreted as "the product of F times H8 is added to J9." However, if you wanted to add H8 and J9 together before multiplying their sum times F, you would have to place parentheses around  $H8 + J9$ :

$$F * (H8 + J9)$$

In this case the addition is performed first because the operation inside the parentheses is carried out before the outside multiplication.

Multiplication and division have equal priority and will be performed from left to right. The same is true of addition and subtraction. If you look at line 40 again

$$40 \ S = P * (1 + R * T)$$

it should be clear that the parentheses are needed. However, inside the parentheses, the expression:

$$1 + R * T$$

is calculated correctly because the multiplication has higher priority than addition.

## Experiment #2 Compound Interest

If you invest  $P$  dollars at an annual interest rate  $r$ , compounded  $k$  times a year, then after  $t$  years, your investment will have grown to the amount  $S$ :

$$S = P(1 + r/k)^{tk}$$

A program will be written to calculate the value of  $S$ , for the following values of the other variables:

$P$  is \$ 5,000

$r$  is 12% per year

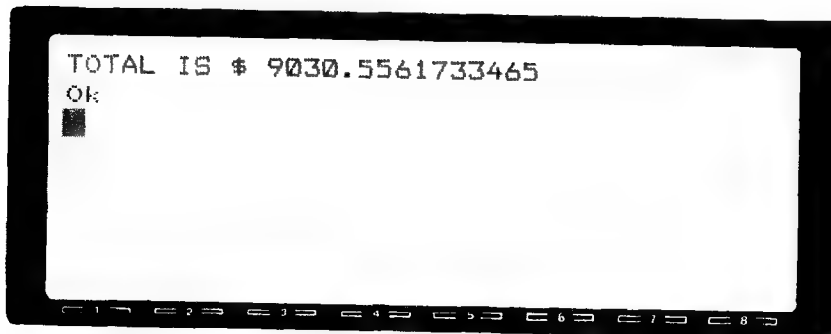
$t$  is 5 years

$k$  is 4 (i.e. interest is compounded 4 times a year.)

Enter the command NEW to clear the previous program from memory. Then type the following program:

```
10 P = 5000
20 R = .12
30 T = 5
40 K = 4
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $" ; S
```

After the program is entered, execute it by entering the RUN command or pressing (F4). The output from this program will be:



Even though the total is supposed to represent dollars and cents, the computer has displayed the number with 14 significant digits, including 10 to the right of the decimal place (because this is the precision of numeric variables in Model 100 BASIC). You will see later how to display numbers in dollars and cents format.

The expression in line 50

$$P * (1 + R / K)^{(T * K)}$$

uses addition, multiplication and exponentiation operators. The exponentiation operator has the highest priority and will be performed before multiplication or division and of course before addition and subtraction. Because of this priority, it was necessary to place parentheses around the exponent

$$T * K$$

Otherwise, the computer would have calculated

$$P * (1 + R / K) ^ T$$

and then multiplied this expression by K.

## Experiment #3 Compound Interest with Keyboard Input

If you wanted to run the compound interest program with different values for the variables, you would have to retype the appropriate lines in the program. Obviously that isn't very practical, specially if you just wanted to figure the interest for several values.



It would be more convenient if you could simply type in the values for the variables as the program is being executed.

The program will be changed so that the value for P, the principal, can be entered during execution. The values for the other variables will be assigned, as before, with assignment statements.

Change line 10 to

```
10 PRINT "ENTER THE PRINCIPAL";
```

and insert line 15

```
15 INPUT P
```

List the program to verify that it is:

```
10 PRINT "ENTER THE PRINCIPAL";
15 INPUT P
20 R = .12
30 T = 5
40 K = 4
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $"; S
```

Execute the program.

When the program executes, the following message (called a “prompt”) will be displayed:

```
ENTER THE PRINCIPAL?
```

and the computer will wait for you to enter the value for P. When a value for P is entered, for example 5000, the program will execute as before and print out the total. Be sure to press **(ENTER)** after typing the amount for P.

Here is what the output looks like:

```
ENTER THE PRINCIPAL? 5000
TOTAL IS $ 9030.5561733465
```

**Line 10** prints the prompt message. The semicolon at the end of the line suppresses the carriage return so that the next character printed will be on the same line.

**Line 15** is the **INPUT** statement. When this statement is executed, a question mark is printed and the computer will wait for you to enter a number from the keyboard. After you type the number and press **(ENTER)**, the number will be assigned to the variable **P** which appears after **INPUT**.

The remainder of the program is the same as before.

If the semicolon is left out in line 10, then the question mark and the number you type will be printed on the next line below the prompt.

## Experiment #4 Another Way to INPUT

The INPUT statement is usually preceded by a prompt message to remind you what value to enter. An optional form of the INPUT statement allows the prompt to be printed without using a separate PRINT statement.

Delete line 15 (by entering just the line number 15) and retype line 10 as follows:

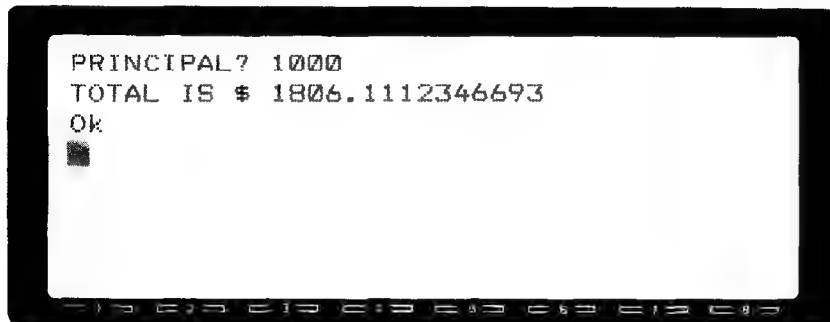
```
10 INPUT "PRINCIPAL"; P
```

This single line is equivalent to the previous lines 10 and 15. The word PRINCIPAL will be printed immediately followed by a question mark. Then the Computer will wait for you to enter the value for P. The ending quotation mark must be followed by a semicolon, a comma cannot be used.

List the program. It should read:

```
10 INPUT "PRINCIPAL"; P
20 R = .12
30 T = 5
40 K = 4
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $"; S
```

Execute the program. The following output occurs when you type 1000 for the principal:



Run the program several times, entering different values for the principal.

## Experiment #5 Compound Interest with More Keyboard Input

In this experiment, the compound interest program will be changed so that the rate, as well as the principal, can be input from the keyboard.

Delete line 20 by entering just the line number.

Retype line 10 as:

```
10 INPUT "PRINCIPAL , RATE"; P , R (ENTER)
```

Now list the program to verify that it is:

```
10 INPUT "PRINCIPAL , RATE"; P , R
30 T = 5
40 K = 4
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $"; S
```

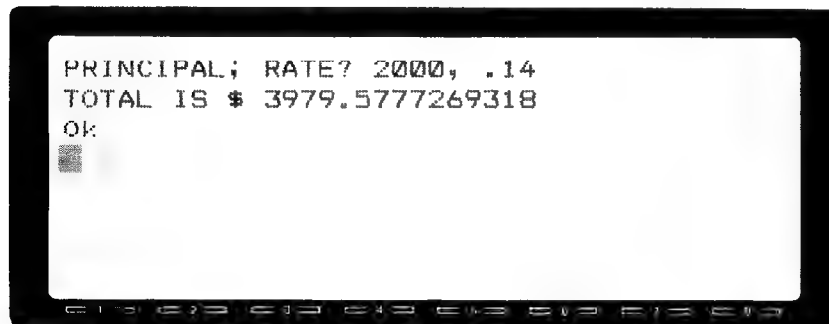
When you execute this program, the following prompt will be displayed

PRINCIPAL , RATE?

In response, you should enter two numbers separated by a comma. For example:

PRINCIPAL ,RATE? 2000 , .14

and again the amount of the investment will be printed. The complete output will be:



This program illustrates another option of the INPUT statement. Values for more than one variable can be assigned with a single INPUT statement. The variables should be separated by commas and listed after the prompt. When the INPUT statement is executed, you must type a value for each variable, separating the values with commas. After the last value has been typed, you should press **(ENTER)**.

Two separate INPUT statements could have been used, the first for the principal P and the second for the rate R as follows

```
10 INPUT "PRINCIPAL"; P
20 INPUT "RATE"; R
```

However, it is more convenient to use just a single statement as was done in the program.

## Experiment #6 Compound Interest with All Variables Input

The previous program will be rewritten so that the values for all the variables can be entered from the keyboard. The program will also be changed so that it will not terminate after the total is printed. Instead, the program will ask again for values for the variables so that the amount of the investment can be calculated for any number of different values.

Delete line 40 by entering just the line number and retype line 30 as follows:

```
30 INPUT "TIME, NUMBER OF PERIODS"; T, K (ENTER)
```

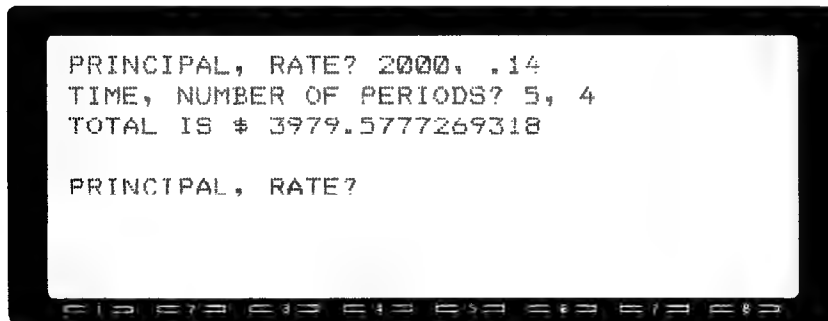
Then add two new lines:

```
70 PRINT (ENTER)
80 GOTO 10 (ENTER)
```

List the program to verify that it is:

```
10 INPUT "PRINCIPAL, RATE"; P, R
30 INPUT "TIME, NUMBER OF PERIODS"; T, K
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $"; S
70 PRINT
80 GOTO 10
```

The program will prompt you for values for all four variables. Here is an example of the output of the program with the following inputs:



```
PRINCIPAL, RATE? 2000, .14
TIME, NUMBER OF PERIODS? 5, 4
TOTAL IS $ 3979.5777269318

PRINCIPAL, RATE?
```

The program will continue to ask for values for the variables until you terminate the program manually by pressing (BREAK).

**Line 10** prints a prompt and allows you to enter values for the principal P and the rate R.

**Line 30** prints a prompt and allows you to enter values for the time T and the number of periods K. Lines 10 and 30 could have been combined into one line if desired.

**Line 60** prints out the message "TOTAL IS \$" and the value of S, the total of principal and interest at the end of T years.

**Line 70** skips a line on the display after the output from line 60. This spaces the output so that it doesn't run together vertically.

**Line 80** transfers control back to line 10, where the program starts over again. This allows you to input a variety of values for the variables without entering the RUN command each time.

## Experiment #7 Personalizing the Output — Printing a Name

The compound interest program as it currently exists is:

```
10 INPUT "PRINCIPAL, RATE"; P, R
30 INPUT "TIME, NUMBER OF PERIODS"; T, K
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL IS $"; S
70 PRINT
80 GOTO 10
```

Add line 40 as follows:

```
40 INPUT "NAME"; N$
```

and change line 60 to

```
60 PRINT "TOTAL FOR "
```

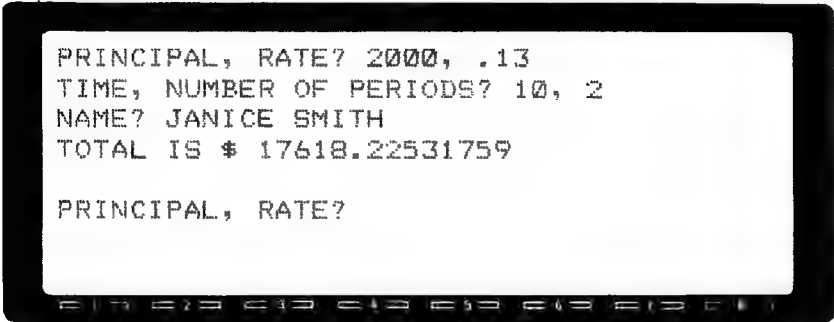
Add line 65

```
65 PRINT N$;" IS $"; S
```

List the program to confirm it is

```
10 INPUT "PRINCIPAL, RATE"; P, R
30 INPUT "TIME, NUMBER OF PERIODS"; T, K
40 INPUT "NAME"; N$
50 S = P * (1 + R / K)^(T * K)
60 PRINT "TOTAL FOR "
65 PRINT N$;" IS $"; S
70 PRINT
80 GOTO 10
```

When you execute the program you will be prompted, as before, for the principal, rate, time and number of periods. In addition, you will be prompted for a name. After this data is entered, the name and the amount of the investment will be printed. Run the program and enter the data as indicated below (of course, you may enter any other name you wish):



```
PRINCIPAL, RATE? 2000, .13
TIME, NUMBER OF PERIODS? 10, 2
NAME? JANICE SMITH
TOTAL IS $ 17618.22531759

PRINCIPAL, RATE?
```

Execute the program several times with various numeric values and names. To terminate execution, **(BREAK)** must be pressed.

Lines 10 - 30 allow you to enter values for the numeric variables in the program.

---

**Line 40** is another INPUT statement. This time, however, you are expected to enter a name instead of a number. The name which you enter is stored in the variable N\$. This variable, N\$, is an example of a "string variable." The name that you type, for example, JANICE SMITH, is called a "character string."

**Line 50** calculates the total of principal and interest.

**Line 60** prints the message "TOTAL FOR."

**Line 65** prints the name stored in N\$ followed by the word IS, the symbol \$, and the amount of the investment.

**Line 70** skips a line (PRINTs a blank line).

**Line 80** transfers control back to line 10.

A **string** is any sequence of keyboard characters, for example

```
JOHN SMITH  
1023 N. MAIN STREET  
1982  
MACROCORP  
MARCH 12, 1984  
$120.95
```

In some cases the string may consist entirely of digits (e.g. 1982), but it can still be considered a string as well as a numerical constant.

A **valid string variable** is any numeric variable followed by a dollar sign (\$). The following are all valid examples of string variables:

N\$, A3\$, DA\$, AD\$, ST\$

In line 40, it is essential that the variable be a string variable. If you tried to use a numeric variable, the computer would continue to ask for data until a numeric constant was entered.

The assignment of the name to the string variable N\$ was made with the INPUT statement. It is possible to use assignment statements with string variables and constants just as with numeric variables and constants. For example, the following statement:

```
100 A$ = "RADIO SHACK"
```

assigns the string RADIO SHACK to the string variable A\$. When using an assignment statement, it is necessary to enclose the string in quotes, as shown above.

Each string variable can hold up to 256 characters. If the total number of characters assigned to string variables exceeds 256, you must set aside additional space. This is done with the CLEAR statement. For example, the following line could be added to the program:

```
5 CLEAR 300
```

This particular instruction allocates space for 300 characters. If you attempt to store more than 300 characters in N\$, an error will result and the program will terminate. When the Computer is turned on, space for 256 characters is allocated automatically.

---

If you require more, then the CLEAR statement must be used. Since it is unlikely a name will be longer than 256 characters, the above program should not need the CLEAR statement.

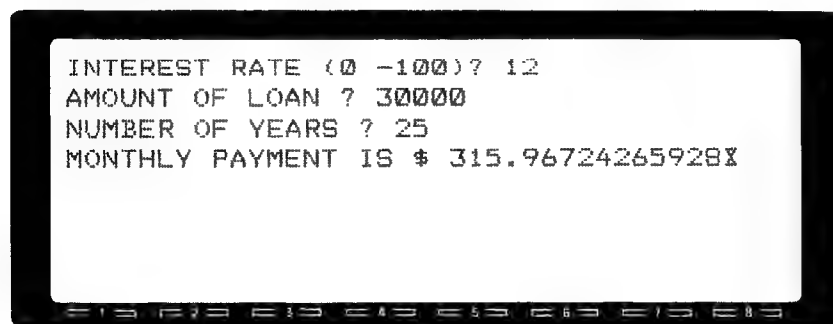
If, on the other hand, your program had twenty string variables, each of which is used to hold a name, then the maximum of 256 would probably be exceeded. In this case, the CLEAR statement would be needed.

## Experiment #8 Mortgage Payment Calculation

Clear working memory with the NEW command and type the following program from the keyboard:

```
10 INPUT "INTEREST RATE (0 - 100)"; R
20 R = R / 100
30 INPUT "AMOUNT OF LOAN "; A
40 INPUT "NUMBER OF YEARS"; T
50 N = 12 * T
60 I = R / 12
70 MP = (1 - (1 + I)^(-N)) / I
80 MP = A / MP
90 PRINT "MONTHLY PAYMENT IS $"; MP
```

When you execute this program, you will be prompted for the interest rate, the amount of the loan and the length in years of the loan. If the interest rate is 12% then 12 should be entered, not .12. Here is an example of the execution of the program.



**Line 10** allows the interest rate to be entered and stores it in variable R.

**Line 20** converts the interest rate to a decimal value.

It is important to note that the equals symbol (=) in a BASIC program means "assign the value computed on the right side to the variable on the left side." It does not mean the right side is equal to the left side as in an algebraic equation. Thus the assignment statement:

$$R = R / 100$$

means to compute the right side,  $R / 100$ , and then store the result back in R.

**Line 30** allows the amount of the loan to be entered and stored in numeric variable A. Remember that numeric values are entered without a comma.

**Line 40** allows the length of the loan (in years) to be entered and stored in variable T.

**Line 50** computes the number of payment periods (months) of the loan and stores the value in variable N.

**Line 60** computes the interest rate per month and stores it in variable I.

**Lines 70 - 80** compute the monthly payment. The computation was done in two lines rather than one, to avoid having a long, complicated expression in a single line.

Recall that the equals symbol (=) means to assign the value computed on the right side to the variable on the left side. In line 80 the value of A/MP is computed and then stored in MP.

**Line 90** prints the amount of the monthly payment.

Run the program several times with your own data. Have you ever wondered what effect a change in the interest rate would have on your mortgage payment? Just run the program and vary the interest rate while keeping the other variables constant.

## Experiment #9 Calculation of Total Amount and Loan Balance

In addition to calculating the monthly mortgage payment, as in the last experiment, you might wish to compute the total amount you have to pay. This is done by multiplying the number of pay periods N by the monthly payment MP. Type in lines 100 and 110 as follows:

```
100 TA = N * MP
110 PRINT "TOTAL AMOUNT PAID IS $"; TA
```

Another calculation that can be made is the determination of the loan balance or outstanding principal, after a certain number of payments. Enter the following lines:

```
120 INPUT "CURRENT YEAR OF LOAN"; Y
130 M = 12 * Y
140 B = (1 - (1 + I)^(M-N)) / I
150 B = MP * B
160 PRINT "PRINCIPAL REMAINING IS $"; B
```

Here is a listing of the the complete program:

```
10 INPUT "INTEREST RATE (0 - 100)"; R
20 R = R / 100
30 INPUT "AMOUNT OF LOAN"; A
40 INPUT "NUMBER OF YEARS"; T
50 N = 12 * T
60 I = R / 12
70 MP = (1 - (1 + I)^(-N)) / I
80 MP = A / MP
90 PRINT "MONTHLY PAYMENT IS $"; MP
100 TA = N * MP
110 PRINT "TOTAL AMOUNT PAID IS $"; TA
120 INPUT "CURRENT YEAR OF LOAN"; Y
130 M = 12 * Y
```

---



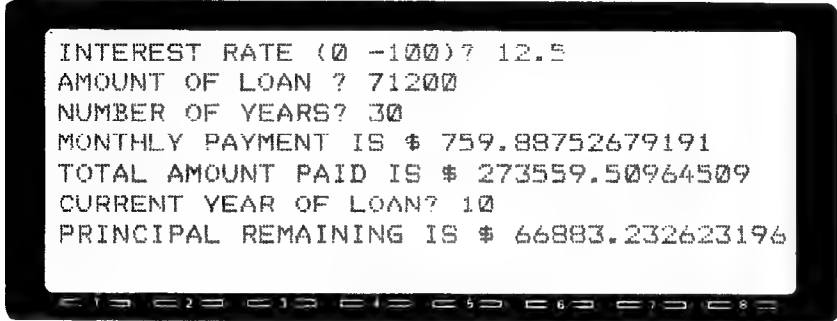
---

```

140 B = (1 - (1 + I)^(M-N)) / I
150 B = MP * B
160 PRINT "PRINCIPAL REMAINING IS $"; B

```

Here is an example of the execution of this program:



```

INTEREST RATE (0 -100)? 12.5
AMOUNT OF LOAN ? 71200
NUMBER OF YEARS? 30
MONTHLY PAYMENT IS $ 759.88752679191
TOTAL AMOUNT PAID IS $ 273559.50964509
CURRENT YEAR OF LOAN? 10
PRINCIPAL REMAINING IS $ 66883.232623196

```

This example would be typical of an \$ 89,000 home purchased with a conventional 80% mortgage at 12.5% over 30 years. The amount of the loan would be \$ 71,200. The discouraging result of running this example is that the total you will have to pay over the 30 years is \$ 273,559.51. By entering a current year of 10, you can see that the principal remaining unpaid after 10 years is still \$ 66,883.23. This means that in 10 years you have paid

$$\text{\$ } 71,200 - \text{\$ } 66,883.23 = \text{\$ } 4,316.77$$

toward the original amount borrowed.

## What you have learned:

In this lesson, the use of string and numeric variables has been illustrated. Arithmetic expressions have been used in the calculation of interest and mortgage rates. The INPUT statement was found to be a very convenient method of entering data into the Computer.



IF/THEN/  
ELSE

PRINT USING

STOP

READ/DATA/  
RESTORE

## Lesson #4 Sales Commissions

It is often necessary to write programs that will do one task if a condition is true and another if the condition is false. This is called **branching**. In this lesson you will learn why this concept is important and how to implement branching in your programs.

### Experiment #1 Sales Commissions

A salesman is to receive a flat rate commission of 15% of his total sales. However, if the total sales is over \$2000, then he will receive an additional 20% of the amount over \$2000.

The formulas are as follows:

Commission = .15 \* sales if sales are less than 2000

Commission = .15 \* sales + .20 \* (sales - 2000) if sales are over 2000

Clear memory with the NEW command and type the following program:

```
10 INPUT "AMOUNT OF SALES"; ST
20 CM = ST * .15
30 IF ST > 2000 THEN CM = CM + .20 * (ST - 2000)
40 PRINT "COMMISSION IS";CM
```

Execute this program.

The program begins by asking you to enter the amount of sales. Type:

1000 (ENTER)

to compute the commission on one thousand dollars sales. Note that a dollar sign is not entered. The computer will respond with the message

COMMISSION IS 150

indicating a commission of \$150.

Since the sales were less than \$2000, the commission is a straight fifteen percent.

RUN the program again and this time when you are asked to enter the amount of sales, type:

3000 (ENTER)

to compute the commission on three thousand dollars sales. The computer will respond with the message:

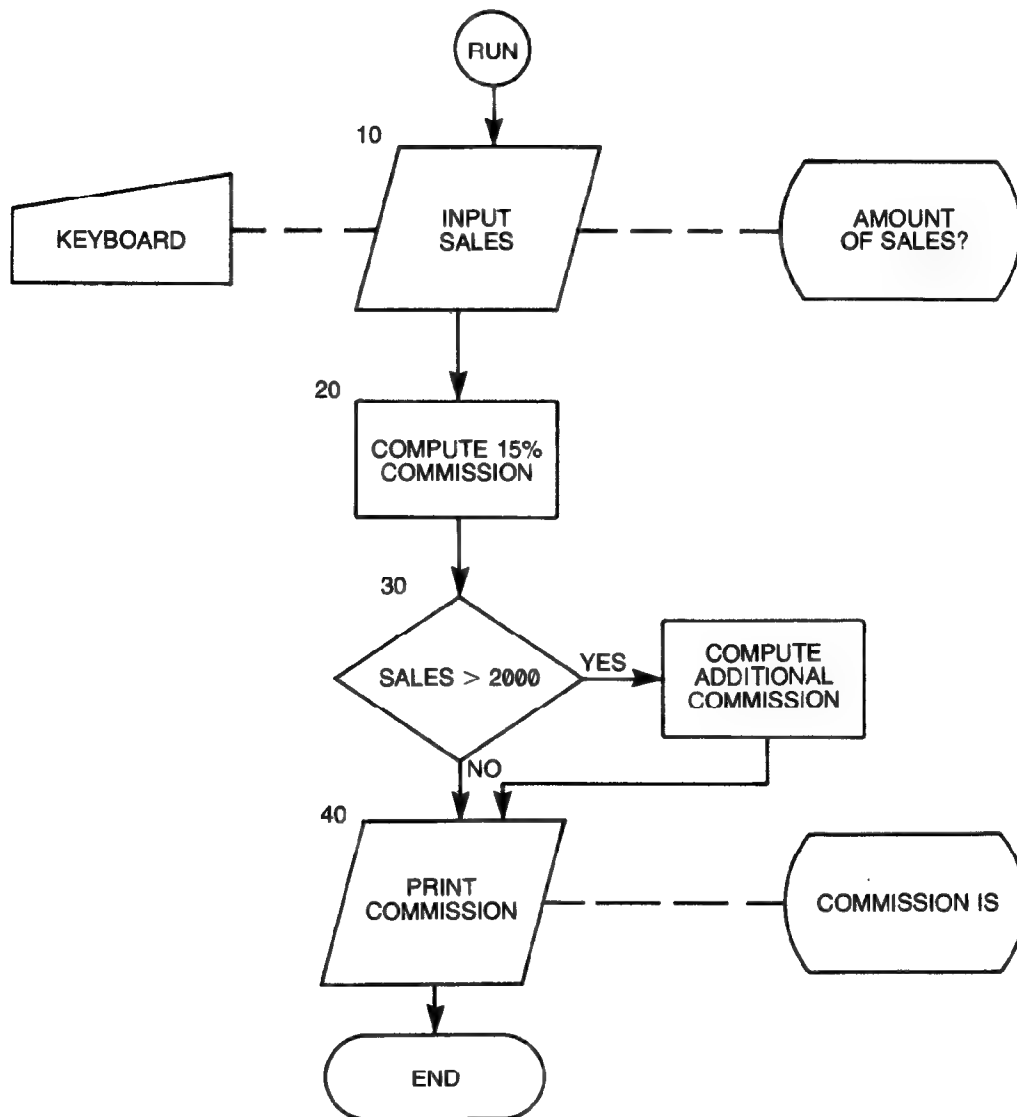
COMMISSION IS 650

indicating a commission of \$650. Since the sales exceed \$2000, the commission will be 15% of \$3000 plus 20% of \$1000 (the amount exceeding \$2000).

Try running the program using your own sales data and confirm that the program computes the commission properly.

---

## How the Sales Commissions Program Works



**Figure 4-1. Flowchart of Sales Commission Program**

When you execute the program, you will be prompted for the total sales. The commission then will be computed according to the above scheme and printed.

**Line 10** allows you to enter the total sales, which is stored in the variable ST.

**Line 20** computes the flat rate sales commission and stores it in the variable CM.

**Line 30** contains a new statement, the **IF/THEN** statement. As can be seen from the flow chart in figure 4-1, the program must branch after line 20.

If the sales exceed \$2000, an additional computation must be performed. However, if this is not the case, then the sales commission is already calculated in line 20 and no additional computation is required.

---

---

To accomplish this, the program must be able to test whether or not sales (ST) is greater than \$2000, and branch accordingly. This can be done with the use of the IF/THEN statement.

The inequality

$$ST > 2000.$$

appearing in line 30 (between IF and THEN) is called a **“condition.”** If this condition is false, then the next statement to be executed is in line 40. On the other hand, if the condition is true, then the statement following THEN:

$$CM = CM + .20 * (ST - 2000)$$

will be executed before control passes on to line 40. In other words, if total sales are not greater than 2000, no additional calculations are performed in line 30. But, if total sales are greater than 2000, the additional commission is calculated and added to the original flat rate commission.

The general format of the IF/THEN statement is

**IF “condition” THEN “statement”**

where **“statement”** is executed only when **“condition”** is true. In either case, the next line executed is the one following the IF/THEN statement.

The condition is usually the comparison of two numeric or string expressions. Two numeric expressions are compared with the use of a **relational operator**. In this case, **“>”** in line 30 is the relational operator.

Here is a table of numeric relational operators:

numeric operator	meaning
>	greater than
<	less than
< > or > <	not equal to
>= or =>	greater than or equal to
<= or =<	less than or equal to

Assuming the variable A has the value 2, the logical values of some conditions are given below.

condition	logical value
A > 1	true
A < - 1	false
A < > 1	true
A >= 2	true
A <= 0	false

Here are some more examples of valid IF/THEN statements

55 IF A \* B - C < D THEN A = B

65 IF 5.6 < > AB - 4.6 THEN STOP

75 IF CD/AC < > 1 THEN CD = AC

Each condition can contain only one relational operator. Thus the following condition is not allowed:

`A < > B < > C`

It should be clear that IF/THEN is a very powerful statement because it allows the program to carry out different tasks and functions, depending upon the value or values of numeric expressions.

## Experiment #2 Printing Dollars and Cents

One unsatisfactory aspect of the Sales Commission program is that the value printed for the commission does not directly indicate dollars in the customary fashion.

Run the program and enter the value 1763.89 for the total sales. You will see that 264.5835 is printed out for the amount of the commission. The output of the program is a decimal number that sometimes contains more than two digits to the right of the decimal point.

It would be much neater if the commission were printed to the nearest cent each time. One way of accomplishing this is to use a different print statement. Instead of using PRINT, the PRINTUSING statement can be used to format the output.

Make the following changes to the program:

Change line 40 as follows:

```
40 PRINT "COMMISSION IS ";
```

and add a new line 50

```
50 PRINTUSING "####.##"; CM
```

Line 40 will display the message

```
COMMISSION IS
```

and line 50 will print the value of the commission. Now if you execute this revised program you will see that the commission is printed, as desired, rounded to the nearest cent.

For example, run the program and enter 1763.89 as the sales amount. This time the commission amount prints as 264.58.

The string enclosed in quotes,

```
####.##
```

is called a **format specifier**, and it indicates how a number is to be printed.

Since two of the # symbols appear to the right of the decimal point, exactly two digits will be printed to the right of the decimal point. By the same token, a maximum of four digits to the left will be printed. (Fewer than four digits to the left of the decimal point may be printed, depending on the size of the number.)

Since there are six # symbols and one decimal point specified, a total of seven columns will be used for the printing of the number. The number will be printed "right justified" in this seven column field, meaning that there may be some blanks in the left most columns, but not in the right-most.

Instead of using a constant string, such as "####.##", it is permissible to use a string variable for the format specifier. For example, the program can be changed as follows

```
45 A$ = "####.##"
50 PRINT USING A$; CM
```

This will give the same output as before. Again, the value printed for the commission will be rounded to the nearest cent. Note that the format specifier, whether it is a string or a string variable, is always followed by a semicolon.

For some more examples, suppose CM = 1416.3812 (actual value as computed) and this is printed out with line 50 above. The following table gives the output for various format specifiers

format specifier	output
A\$ = "####.##"	1416.38
A\$ = "####.#"	1416.4
A\$ = "#####.###"	1416.381
A\$ = "#####.####"	1416.38120

## Experiment #3 Dollar Signs

There are other possibilities for format specifiers. Change line 45 in the program so that it is:

```
45 A$ = "$#####.##"
```

Now run the program and input a variety of values for the sales. In each case there will be a dollar sign (\$) printed to the left of the field (9 column). In many cases there will be some blanks between the \$ and the left most digit of the number.

Now change line 45 so that it is:

```
45 A$ = "$$#####.##"
```

Run the program with a sales amount of 1111. As you can see, the \$ is now printed just to the left of the number \$166.65. The use of two dollar signs to the left of the numeric field specifier instructs the computer to print a dollar sign immediately to the left of the leading digit.

## Experiment #4 Checks

Many times, payroll checks are printed with asterisks padding the leftmost columns of the field.

Change line 45 in the previous program to:

```
45 A$ = "***$#####.##"
```

Run the program with a sales amount of 1111. In this case you can see that in the output of the commission

```
*****$166.65
```

the unused positions are, indeed, padded with the "\*" symbol.

Large dollar amounts are usually printed with commas. Try changing line 45 to

```
45 A$ = "****$,***,***"
```

Run the program with a sales amount of 123456. In the output of the commission,

```
***$42,809.60
```

the digits are now separated with a comma to make the reading of the number easier.

If your number requires more columns than you have specified, for example printing 34.56 with "#.###", the number will be printed anyway, but the symbol "%" will be printed to the left of the number to indicate the field overflow.

## Experiment #5 Sales Commission Revisited

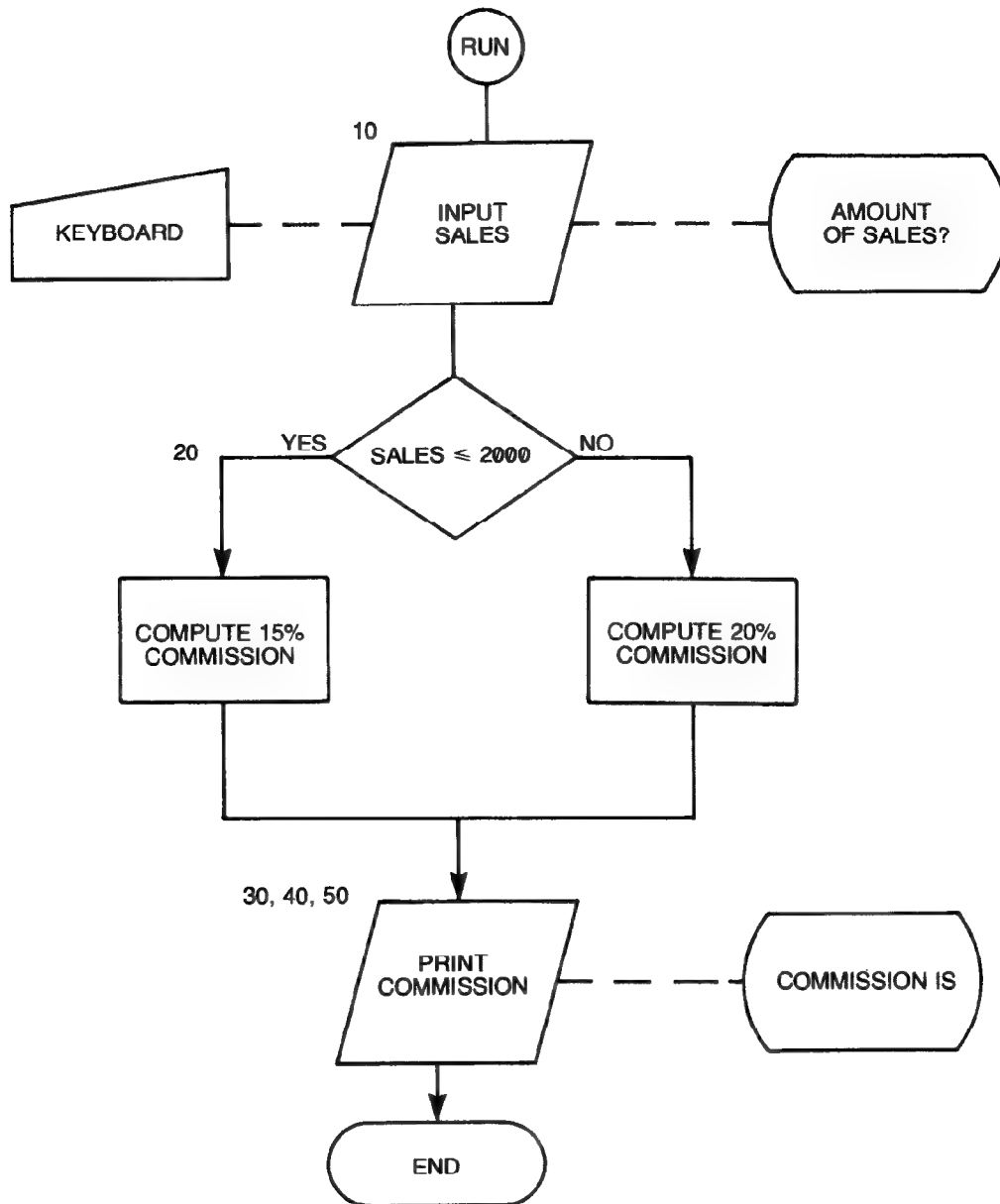
Change the sales commission program so that it is computed according to the following scheme:

- if sales are \$2000 or under, commission is 15% of sales
- if sales are over \$2000, commission is 20% of sales

The program must now make two different computations depending upon the total sales.

The program is illustrated with the flow chart in figure 4-2. The branching can be accomplished with the use of a variation of the IF/THEN statement, called the IF/THEN/ELSE statement.





**Figure 4-2. Flowchart of Experiment 5**

Modify the program by deleting line 20 and replacing line 30:

```

30 IF ST <= 2000 THEN CM = .15 * ST ELSE CM
   = .20 * ST

```

Since this line requires more than 40 columns, it will not fit on a single line of the display of your Model 100, but will overflow to the next line. A line in your program may contain up to 255 characters, in which case it will extend over several lines of the display.

A line is terminated only when you press **(ENTER)** and not at the end of a display line.

List the program to confirm that it is:

```
10 INPUT "AMOUNT OF SALES"; ST
20 IF ST <= 2000 THEN CM = .15 * ST ELSE CM = .20 * ST
40 PRINT "COMMISSION IS";
45 A$ = "***$***,***.***"
50 PRINT USING A$ ; CM
```

Execute the program.

When prompted for the amount of sales, enter 1000. The commission will be computed as \$150.00, which is a straight 15%.

Run the program again and enter a sales of 3000. This time the commission will be computed as \$600.00 which is a straight 20%.

The IF/THEN/ELSE statement is used to execute one of two different statements depending upon the logical value of the condition.

If sales are less than or equal to 2000, the commission is calculated as 15% of sales and control passes to line 40. However, if sales are over 2000, then the statement following ELSE is executed instead, so that the commission is calculated as 20% of sales and again control passes to line 40.

The general format of the IF/THEN/ELSE statement is:

**IF "condition" THEN "statement 1" ELSE "statement 2"**

"statement 1" is executed if "condition" is true, but "statement 2" is executed if "condition" is false

In either case, control passes to the next line in the program. The statements executed can be any valid BASIC statements. Here are some more valid examples of the IF/THEN/ELSE statement:

```
100 IF A = 0 THEN A = B + 1 ELSE A = B - 1
200 IF A + B < 100 THEN A = B ELSE A = 0
```

## Experiment #6 Computing Commissions for Several Salesmen

The Commission Sales program, as it currently exists, will compute the commission for one salesman and terminate. To compute the commissions for several salesmen, it must be modified.

List the program to confirm that it still looks like this:

```
10 INPUT "AMOUNT OF SALES"; ST
20 IF ST <= 2000 THEN CM = .15 * ST ELSE
   CM = .20 * ST
40 PRINT "COMMISSION IS";
45 A$ = "***$***,***.***"
50 PRINT USING A$ ; CM
```

Line 10 will be changed so that the name of the salesman can be entered along with the sales amount.

```
10 INPUT "NAME, SALES"; N$, ST
```

The name will be placed in the string variable N\$ and the sales amount in the numeric variable ST. Line 20 will correctly calculate the commission, but line 40 must be altered so that the name is printed. Change line 40 to:

```
40 PRINT "COMMISSION FOR "; N$;" IS ";
```

If the name entered is SMITH, then line 40 will print

```
COMMISSION FOR SMITH IS
```

and line 50 will print the amount of the commission after the word IS.

Here is a listing of the program after the changes have been made:

```
10 INPUT "NAME, SALES"; N$, ST
20 IF ST <= 2000 THEN CM = .15 * ST ELSE
   CM = .20 * ST
40 PRINT "COMMISSION FOR "; N$;" IS ";
45 A$ = "*****,**.***"
50 PRINT USING A$; CM
```

If this program were executed, it would compute the commission of a single salesman and execution would terminate. If the program is to compute more than one sales commission, it must branch back to line 10. This can be accomplished with the use of a GOTO statement after line 50. Type in a new line:

```
60 GOTO 10
```

Now the program is complete:

```
10 INPUT "NAME, SALES"; N$, ST
20 IF ST <= 2000 THEN CM = .15 * ST ELSE
   CM = .20 * ST
40 PRINT "COMMISSION FOR "; N$;" IS ";
45 A$ = "*****,**.***"
50 PRINT USING A$; CM
60 GOTO 10
```

This program illustrates a concept in programming called "looping." This means that a block of statements are repeated several or perhaps many times in a program. In the program above, the entire program is repeated, each time with a different salesman and sales total. It should be noted that the program is an "infinite loop," which means it will not terminate and must be manually terminated by pressing **(BREAK)**.

Try running the program and entering several names and sales amounts. Be sure to press **(BREAK)** when you wish to terminate execution.

## Experiment #7 How to Escape From an Infinite Loop

It is easy to add some statements so that programs that contain infinite loops will terminate upon request.

In the Sales Commission program, for example, the sales total will never be negative. But you can modify the program so that it terminates whenever a negative value for sales is entered.

Type the new line:

```
15 IF ST < 0 THEN STOP (ENTER)
```

The **STOP** statement, terminates execution and has the same effect as pressing **(BREAK)**. The program will terminate if ST is negative, but continue if it is not.

Run the program.

When prompted to do so, enter a dummy name (such as END) and a negative sales amount (such as -1). Note that the program terminates immediately without computing any commission.

The dummy name was necessary because the **INPUT** statement in Line 10 requires you to enter two input quantities. Since a negative sales amount terminates execution, the name you enter will not be used.

You can have more than one **STOP** statement in your program. Of course your program need not have any **STOP** statements in it at all.

## Experiment #8 Individual Commissions for the Salesmen

The Sales Commission program calculates every salesman's commission using the same rate. Now we will modify the program so that each salesman has his own commission rate. Here is a list of salesmen and their commission rates:

Name	Rate
ADAMS	15%
JONES	16%
LEE	18%
SMITH	20%
VINSON	14%

When any of the above names is entered, the program must look in the table to find the corresponding rate before the commission can be calculated. Make the following changes to the program:

Retype line 20 as follows

```
20 READ N1$, CR
```

and add lines 24, 26 and 28

```
24 IF N1$ <> N$ THEN GOTO 20
26 CM = ST * CR
28 RESTORE
```

Delete line 30.

Now the only remaining change is to add the table to the program. Type the following line:

```
70 DATA ADAMS,.15,JONES,.16,LEE,.18,SMITH,.20,
VINSON,.14
```

List the program to confirm that you have:

```
10 INPUT "NAME, SALES"; N$, ST
15 IF ST < 0 THEN STOP
20 READ N1$, CR
24 IF N1$ <> N$ THEN GOTO 20
26 CM = CR * ST
28 RESTORE
40 PRINT "COMMISSION FOR "; N$; " IS "
45 A$ = "***,***,***,***"
50 PRINT USING A$; CM
60 GOTO 10
70 DATA ADAMS,.15,JONES,.16,LEE,.18,SMITH,.20,
VINSON,.14
```

Run this program.

Be sure to enter one of the five names listed in the table. The program will compute and print the commission for that salesman. Continue to input names in any order and their corresponding sales totals. To end the program enter a dummy name and negative value for the sales amount.

**Line 20** reads the first name (ADAMS) listed in the **DATA** statement in line 70 and places it in the string variable N1\$. It reads the first commission rate (.15) and places it in the variable CR.

**Line 24** compares the name in N\$, which was entered in line 10, to that in N1\$. If they are different, the program jumps back to line 20 which then reads the next name and commission rate in the **DATA** statement. The program continues looping in this way until a match is found. When this happens, CR will contain the correct commission rate for the salesman.

**Line 26** computes the commission for the salesman.

**Line 28** is the **RESTORE** statement. When this statement is executed, the computer will go back to the first item in the data list. Therefore the next time a **READ** statement is executed, it will use the first item in the **DATA** list.

Without **RESTORE**, the reading of the data would continue where it left off from the time before. If the names are entered in an arbitrary order, the data list must be read from the beginning each time.

The remainder of the program is the same as before, with the exception of line 70. Line 70 contains the **DATA** statement from which the names and rates are read. This

statement can be placed anywhere in the program, but it is traditional to place it either at the beginning or the end of the program.

What would happen if a name is entered, which is not in our list?

Run the program and enter the name CARTER and a sales amount of 1000. The program continues reading names in the DATA list until it runs out of data. The program terminates and displays the error message:

```
?OD ERROR IN 20
```

indicating that an OUT OF DATA error occurred in line 20. The program tried to read past the last name and rate in the list.

Additional names and rates can be added to line 70 or placed in another DATA statement. The computer treats multiple DATA statements as one continuous list.

## **What you have learned:**

In this lesson you have learned how a program can be made to **BRANCH** and accomplish different tasks depending upon the value of a variable or expression. This can be done by using the **IF/THEN** statement. The **PRINTUSING** statement was used to format the output so that it was presented in a more reasonable fashion than would be the case with just the **PRINT** statement. You saw how to use the **STOP** statement to terminate execution under program control. You also learned that the **READ/DATA** statements often provide a convenient method of inputting data to the program, especially frequently referenced data like tables and lists.

---

## Lesson #5 Day, Time and Date

Your Model 100 has a number of string functions which allow you to manipulate string constants and string variables in various ways.

In this lesson you will learn how to use some special string functions that "return" the day, time and date. Also, other string functions will be used to extract information from string constants and string variables and to output character data.

### Experiment #1 What Day is it?

DAY\$ is a string function which returns the day of the week. If you type:

```
PRINT DAY$ (ENTER)
```

you will see that the first three letters of the present day are displayed. For example, if today happens to be Thursday, the Computer will display

```
Thu
```

**Note:** You must initialize the day sometime prior to using the DAY\$ function. Once initialized, the day will be automatically updated.

If you have not already initialized the day, this can be done by entering the following command:

```
DAY$ = 'xxx'
```

where "xxx" are the first three letters of the current day. For example, if today is Monday, enter

```
DAY$ = "MON" (ENTER)
```

While the DAY\$ function returns the first three letters of the day of the week, it is often desirable to print the full name of the day (e.g. Thursday, not Thu).

The following program, which we'll call DAY, accomplishes this. Refer to Figure 5-1 for a flowchart of the program.

Clear working memory with the NEW command and type the following program in from the keyboard.

```
10 READ D$
20 IF LEFT$(D$,3) = DAY$ GOTO 40
30 GOTO 10
40 PRINT "TODAY IS "; D$
50 DATA Sunday, Monday, Tuesday, Wednesday
60 DATA Thursday, Friday, Saturday
```

Execute this program.

The program will output the day of the week. Here is an example of the output:

```
TODAY IS Thursday
```

DAY\$

TIME\$

DATE\$

STR\$

STRING\$

VAL

LEFT\$

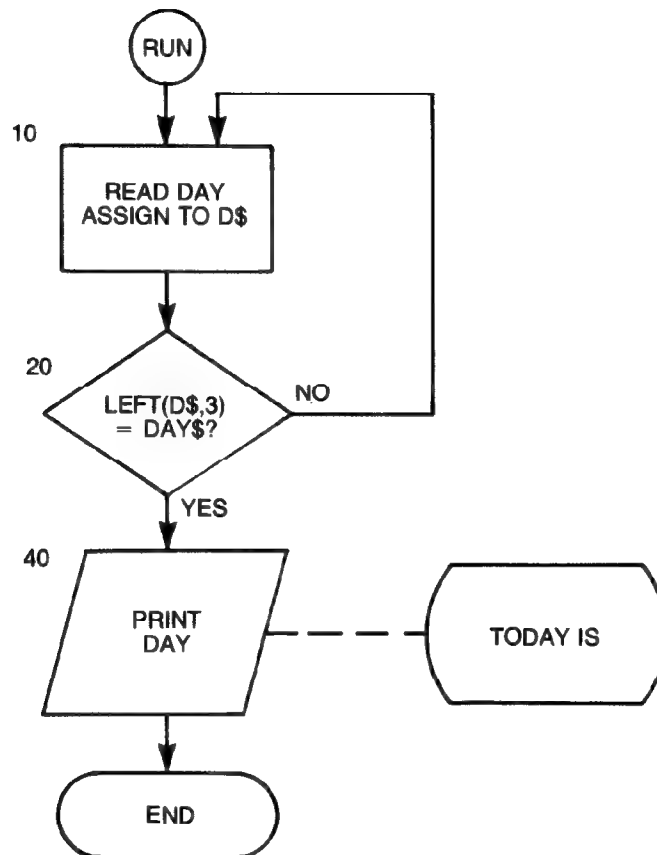
MID\$

RIGHT\$

LEN

---

## How program DAY works:



**Figure 5-1. Flowchart of Program DAY**

**Line 10** A string from line 60 or 70 is read and assigned to the string variable D\$. The first time line 10 is executed, the string Sunday will be assigned to D\$, the second time line 10 is executed, Monday will be assigned to D\$, etc.

**Line 20** The IF / THEN statement compares DAY\$ with the first three characters stored in D\$. DAY\$ contains the first three characters of the current day (SUN, MON, etc.). LEFT\$(D\$,3) returns the three leftmost letters of the string stored in D\$. If a match occurs, the program jumps to line 40, otherwise line 30 is executed next.

*Note that the IF statement does not contain the keyword THEN. The keyword is optional in an IF statement when it is followed by a GOTO statement.*

**Line 30** The GOTO statement transfers control back to line 10 where the next day will be read from the data list.

**Line 40** The PRINT statement displays the message "TODAY IS" followed by the day of the week.



---

**Lines 50 - 60** These DATA statements contain the list of the days of the week. Lowercase letters must be used for the second and third characters to correctly match DAY\$, which stores the first three characters of each day of the week (an uppercase and two lowercase letters).

This program uses two string functions, DAY\$ and LEFT\$. Recall that DAY\$ returns the first three letters of the day of the week.

The function LEFT\$(D\$,3) returns the first three characters of the string stored in D\$. For example, if D\$ has Thursday assigned to it, then

```
PRINT LEFT$(D$,3)
```

would display

```
Thu
```

The number 3 indicates the number of characters to be returned. Again assuming that D\$ contains Thursday, then

Command	Displays as
PRINT LEFT\$(D\$,1)	T
PRINT LEFT\$(D\$,2)	Th
PRINT LEFT\$(D\$,5)	Thurs
etc.	

LEFT\$ is a string function of two arguments, the string (D\$) and the number of characters to be returned. The first argument need not be a string variable, it can also be a string constant. For example,

```
PRINT LEFT$("Model 100",3)
```

would display as:

```
Mod
```

When a string constant is used as the argument, it must be enclosed in quotation marks.

The last character in the name of both functions, LEFT\$ and DAY\$, is a dollar sign (\$) because the quantity returned, in both cases, is a string.

## Experiment #2 Centering the Day

Program DAY is to be changed so that the output will consist of the day of the week, centered in the line with an equal number of asterisks (\*) printed on both sides.

Make the following changes:

Retype line 40 as:

```
40 L = LEN(D$)
```

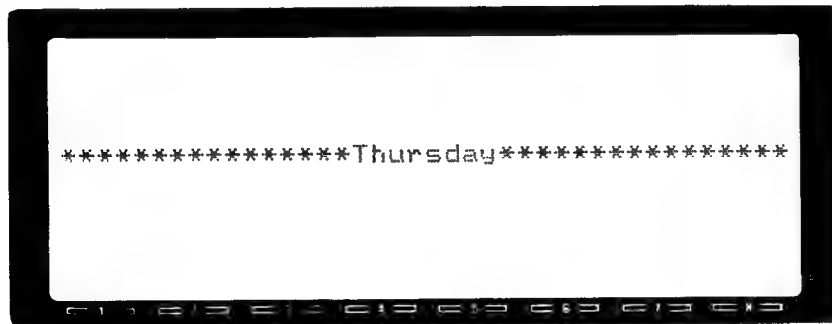
and type four new lines:

```
5  CLEAR 100
44 A$ = STRING$( (40 - L)/2, "*")
46 D$ = A$ + D$ + A$
48 PRINT D$
```

Now list the program to confirm that it is:

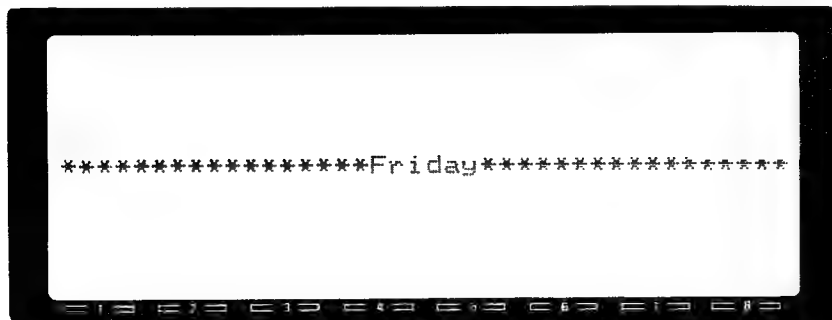
```
5  CLEAR 100
10 READ D$
20 IF LEFT$(D$,3) = DAY$ THEN GOTO 40
30 GOTO 10
40 L = LEN(D$)
44 A$ = STRING$( (40 - L)/2, "*")
46 D$ = A$ + D$ + A$
48 PRINT D$
50 DATA Sunday, Monday, Tuesday, Wednesday
60 DATA Thursday, Friday, Saturday
```

Execute the program by entering the RUN command. If the current day is Thursday, the following will be output:



\*\*\*\*\*Thursday\*\*\*\*\*

If the current day is Friday, then the output will appear as:



\*\*\*\*\*Friday\*\*\*\*\*

In each case, the word is centered in the line and an equal number of asterisks are printed on both sides to fill the line. If the current day is Tuesday, which has 7 characters, then 16 asterisks are printed on both sides so that 39 columns are used.

The function **LEN** returns the number of characters in a string variable or string constant. For example if D\$ contains Friday, then the function **LEN(D\$)** returns the value 6, and **LEN("Friday")** also returns a 6.

Since the value returned by the function **LEN** is an integer, there is no dollar sign (\$) attached to the function name. In line 40, the number of characters of the string stored in D\$ is assigned to L. If the current day is Friday, then L has 6 assigned to it.

Since each line has 40 print columns, the number of asterisks to be printed on each side of the day is

$$(40 - L)/2.$$

For example, if the current day is Friday, 17 asterisks must be printed before and after the word Friday.

In line 44, the function **STRING\$** is used to construct a string of asterisks of the correct length and assign it to the variable A\$. Again, if L has the value 6, then

$$(40 - L)/2 = 17$$

and **STRING\$( (40 - L)/2, "\*" )** will be a string of 17 asterisks. If the current day is Tuesday, then L = 7 and

$$(40 - L)/2 = 16.5$$

This value will be truncated to 16 in the **STRING\$** function so that A\$ will have length 16.

The general form of the **STRING\$** function is

**STRING\$(r, "x")**

which constructs a string consisting of *r* repetitions of the character *x*.

The argument *r* can be a numeric constant, variable or expression. For example, if J=3 and M=2

Command	Displays as
PRINT STRING\$(5, "#")	#####
PRINT STRING\$(J, "X")	XXX
PRINT STRING\$(J-M, "&")	&

Line 46 illustrates the concatenation operator +. The strings A\$, D\$ and A\$ are joined together to form a new string which is then assigned to the variable D\$. The concatenation operator can be used with string constants or string variables. For example:

A\$ = "RADIO "+"SHACK"

concatenates the two strings

"RADIO " and "SHACK"

together to form a new string

"RADIO SHACK"

and assigns it to the string variable A\$.

Note that in line 46, D\$ appears both on the left and the right side of the equal sign. Thus the old value which was assigned to D\$ is replaced by the new string, which is printed in line 48.

## Experiment #3 What time is it?

Another useful feature of your Model 100 Computer is the string function TIME\$, which returns the time as a string of the form

hh:mm:ss

For example, if the time is exactly 1:05 PM, then the following command:

```
PRINT TIME$
```

would print

13:05:00

The first two digits indicate the hour (1 PM), the second two digits the minutes, (5 minutes after 1), and the last two digits the seconds.

The clock is a 24 hour clock, so the hours will range from 00 to 23, with 00 indicating midnight.

**Note:** The time must be initialized prior to using the TIME\$ function. Once set, it will keep the correct time just like a clock. If you have not already initialized the time, you may do so by entering the following command:

```
TIME$ = "hh:mm:ss"
```

where *hh* is the hour, *mm* the minute, and *ss* the seconds. The hour is in 24 hour format, so that 3 PM is hour 15. For example, if the current time is 3:05 PM, type

```
TIME$ = "15:05:00" (ENTER)
```

The following program will convert the string returned by TIME\$, for example 14:18:00, to the customary format, 2:18 PM.

Clear the previous program from memory with the NEW command and then type the following program:

```
10 HH$ = LEFT$(TIME$,2)
20 HH = VAL(HH$)
30 IF HH >= 12 THEN A$ = " PM" ELSE A$ = " AM"
40 IF HH > 12 THEN HH = HH - 12
50 HH$ = STR$(HH)
60 MM$ = MID$(TIME$,4,2)
70 T$ = HH$ + ":" + MM$ + A$
80 PRINT "THE TIME IS"; T$
```

After the program has been entered, execute it with the RUN command. An example of the output is

```
THE TIME IS 1:05 PM
```

**Line 10** The first two characters in TIME\$ are assigned to the string variable HH\$. These two characters designate the hour. For example, if it is 1:05 PM, then HH\$ will contain the two characters 13. Since LEFT\$ returns a string, these two characters form a string even though they are numerical digits. Thus they cannot be assigned to a numeric variable.

**Line 20** The string assigned to HH\$ is converted to a numerical value by the VAL function. This numerical value is assigned to the numeric variable HH. Using the previous example, HH would have the number 13 assigned to it. Without the VAL function, it would not be possible to make this assignment.

**Line 30** This statement determines whether it is AM or PM and assigns an appropriate string value to A\$. The value stored in HH is compared to the numeric constant 12. Only a numeric variable can be used in this way. The use of HH\$ would have produced an error and it was for this reason that the string stored in HH\$ was changed to a numeric constant and assigned to the numeric variable HH.

**Line 40** The hour value is converted from a 24 hour format to a 12 hour format. For example, hour 13 would be converted correctly to 1 (PM). In this line, as well in the previous line, the variable must be a numeric variable. The statement:

HH\$ = HH\$ - 12

is illegal and would produce an error.

**Line 50** This line uses the STR\$ function to convert the numeric value stored in HH back to a string which is then assigned back to the string variable HH\$. This will be concatenated later with another string. In order to do this, a string variable must be used.

**Line 60** This line uses the MID\$ function to read the two minutes digits from TIMES\$ and stores this two-character string in the string variable MM\$. Note that the minutes are the 4th and 5th characters in the string:

13:05:00

MID\$(TIMES\$,4,2) returns two characters starting with the 4th from the left.

**Line 70** In this line, the string to be printed is concatenated together from the variables HH\$ (hour), MM\$ (minutes), and A\$ (AM or PM). A colon is inserted between the hour and minutes.

**Line 80** The message "THE TIME IS" is displayed followed by the time.

Here are some examples of TIMES\$ and the resulting output of the program:

<u>TIMES\$</u>	<u>OUTPUT</u>
10:15:12	THE TIME IS 10:15 AM
17:08:55	THE TIME IS 5:08 PM
03:18:43	THE TIME IS 3:18 AM

Note that a leading zero is not printed for the hour. This leading zero is lost when the VAL function is applied.

The function VAL converts a string constant or variable to a numeric value. The following examples illustrate this function:

<u>string</u>	<u>VAL</u>
123ABBA	123
2.3CA987	2.3
567.4	567.4

The leftmost characters, up to the first character which cannot be part of a number, are converted to a number. This function is useful when it is necessary to use a

numeric string constant in a numeric expression. The number in string form must be converted to a numeric constant before it can be used in a numeric expression.

The STR\$ function is the inverse of the VAL function. It converts a numeric constant or variable to its string form. This is useful when you want to concatenate the number with another string as was done in this program.

The MID\$(string, p, n) returns the sub-string of length n starting with the p-th character. This is illustrated with the following examples:

string	p	n	MID\$(string,p,n)
ABCDEFGH	5	3	EFG
JUNE 18, 1983	10	2	19
RADIO SHACK	7	5	SHACK

## Experiment #4 Printing the Seconds

The program in Experiment 3 did not print the seconds even though they are returned by TIME\$. The program will be changed so that the output will appear as:

THE TIME IS 1:05 PM AND 28 SECONDS

Make the following changes to the program:

Type a new line:

```
65 SS$ = RIGHT$(TIME$,2)
```

and change line 70 to

```
70 T$ = HH$ + ":" + MM$ + A$ + " AND " +  
    SS$ + " SECONDS"
```

List the program to confirm that it is:

```
10 HH$ = LEFT$(TIME$,2)  
20 HH = VAL(HH$)  
30 IF HH >= 12 THEN A$ = " PM" ELSE A$ = " AM"  
40 IF HH > 12 THEN HH = HH - 12  
50 HH$ = STR$(HH)  
60 MM$ = MID$(TIME$,4,2)  
65 SS$ = RIGHT$(TIME$,2)  
70 T$ = HH$ + ":" + MM$ + A$ + " AND " + SS$  
    + " SECONDS"  
80 PRINT "THE TIME IS"; T$
```

Execute this program.

The output will depend on the time. If TIME\$ = 17:23:56, then the following will be displayed:

THE TIME IS 5:23 PM AND 56 SECONDS

Line 65 The RIGHT\$ function is used to return the two rightmost characters stored in TIME\$. These characters are the seconds and are assigned to the string variable SS\$. The RIGHT\$ function is similar to LEFT\$, except that the digits are counted from the

right instead of the left of the string. Of course, the MID\$ function could have been used instead. Using MID\$, line 65 would appear as:

```
65 SS$ = MID$(TIME$,7,2)
```

It is usually easier to use LEFT\$ or RIGHT\$ instead of MID\$ if the characters are the leftmost or rightmost of the string.

## Experiment #5 What's the date?

The string function DATE\$ returns the date in the form

mm/dd/yy

For example, if the present date is October 23, 1983, then the command:

```
PRINT DATE$
```

will display

```
10/23/83
```

**Note:** The date must be initialized prior to using the DATE\$ function. Once entered, the date will be automatically updated. If you have not already initialized the date, you may do so by entering the following command:

```
DATE$ = "mm/dd/yy"
```

where *mm* is the number of the current month (e.g. 03 for March), *dd* is the day, and *yy* the year. For example, if today is December 25, 1983, enter

```
DATE$ = "12/25/83"
```

Delete the previous program from memory with the NEW command. The following program will print the date in the usual form, (i.e., the name of the month, the day and the year.) For example:

```
DECEMBER 25, 1983
```

Type the following program:

```
10 MM$ = LEFT$(DATE$, 2)
20 MM = VAL(MM$)
30 READ MM$
40 CT = CT + 1
50 IF CT < MM THEN GOTO 30
60 DD$ = MID$(DATE$, 4, 2)
70 YY$ = RIGHT$(DATE$, 2)
80 PRINT MM$; " "; DD$; ", 19"; YY$
90 DATA JANUARY, FEBRUARY, MARCH, APRIL
100 DATA MAY, JUNE, JULY, AUGUST, SEPTEMBER
110 DATA OCTOBER, NOVEMBER, DECEMBER
```

Refer to Figure 5-2 for the flowchart of this program.

Execute the program. The output will be today's date in the usual format: month, day, and year.

# How Program "What's the date?" Works

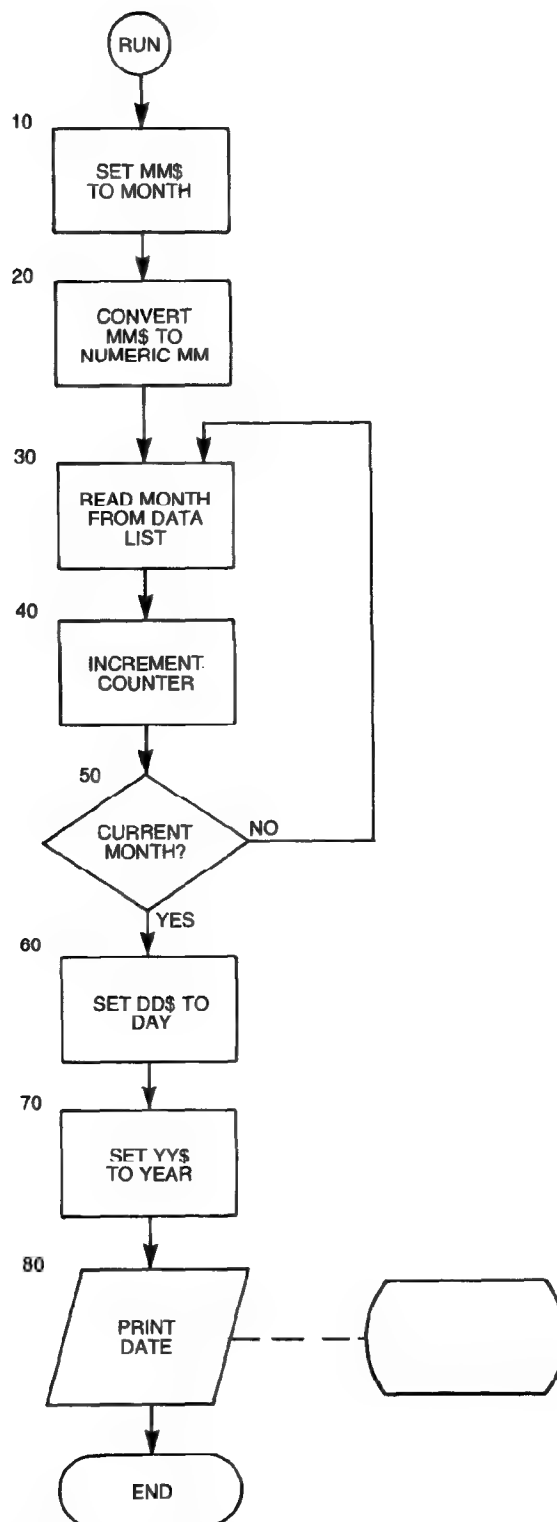


Figure 5-2. Flowchart of Program "What's the date?"



**Line 10** The left two characters of DATE\$ are stored in the string variable MM\$. This is the month. For example, if the month is OCTOBER, then MM\$ will have the string "10" assigned to it. It is important to remember that LEFT\$ returns a string, not a numeric constant.

**Line 20** The string stored in MM\$ is converted to a numeric constant by the VAL function. This must be done because it will be necessary to compare the month number to another numeric constant in line 50.

**Line 30** The next month is read from the DATA statements and placed in MM\$. The first time, JANUARY is assigned to MM\$. Since the string originally stored in MM\$ will no longer be needed, the string variable MM\$ is reused in this line.

**Line 40** This line increments a counter CT. After the first month is read, CT will have the value 1 because the computer initializes all numeric variables to zero when the program is executed.

**Line 50** If the counter CT is less than the number of the current month, then the program will jump back to line 30, where the next month is read from the data list. Eventually, CT will equal MM. When this happens, MM\$ will have the current month assigned to it and line 60 will be executed next. Thus, the months are read repeatedly into MM\$, until the present month is reached, and then the program jumps out of the loop (lines 30, 40 and 50).

**Line 60** The MID\$ function is used to extract the day from DATE\$. The day is given by the 4th and 5th characters of the string stored in DATE\$.

**Line 70** The RIGHT\$ function is used to obtain the current year from DATE\$. Recall that the year is given by the last two characters.

**Line 80** The date is printed in this line. Note that each item in the print list is followed by a semicolon which means that no columns are to be skipped. It was necessary to print a blank space after the month, or otherwise the day would be printed immediately after the month.

**Lines 90 - 110** These DATA statements contain the months of the year which are read by the READ statement in line 30.

## What you have learned:

In this lesson the three special string functions DAY\$, TIME\$, DATE\$ and some of their uses have been illustrated. Various other string functions have been used to extract information from string variables and to print out certain string quantities.



## Lesson #6 Using the Editor

In this lesson you will learn how to use the built-in **Editor** of the Model 100 so that changes to a BASIC program can be made quickly and easily.

Until now, you have been able to make changes to your BASIC programs in the following three ways:

- 1) An existing line is changed by retyping it.
- 2) An existing line is deleted by entering just the line number.
- 3) A new line is added by entering it with the appropriate line number.

While any change to your program can be accomplished using these three procedures, they can be time consuming. For example, if you only want to change a single character in a line of your program, the entire line must be retyped. If you want to move one line to another location in the program, the old line must be deleted and the line retyped with a new line number.

An easier and more efficient way to make changes of this type is to use the built-in Editor. The Editor allows changes to be made to a line without retyping the entire line. It also allows a line number to be changed without retyping the line.

There are other convenient features of the Editor as well. These features will be examined in detail.

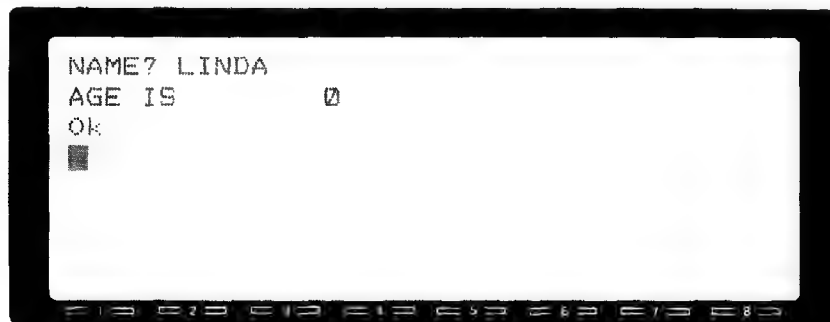
### Experiment #1 Inserting a Character

Enter the following program from the keyboard exactly as it is printed here:

```
10 INPUT "NAME"; N$
20 READ A$, AG
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "AGE IS", A
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

This program prompts you for a name. If one of the names in the DATA statements is entered, (i.e. BETTY), the corresponding age (35) will be displayed. If the name entered is not found in the DATA statements, an out of data error will result (at least that is how the program is supposed to work).

Run the program. If you enter the name LINDA, the output will look like this:



```
NAME? LINDA
AGE IS      0
Ok
█
```

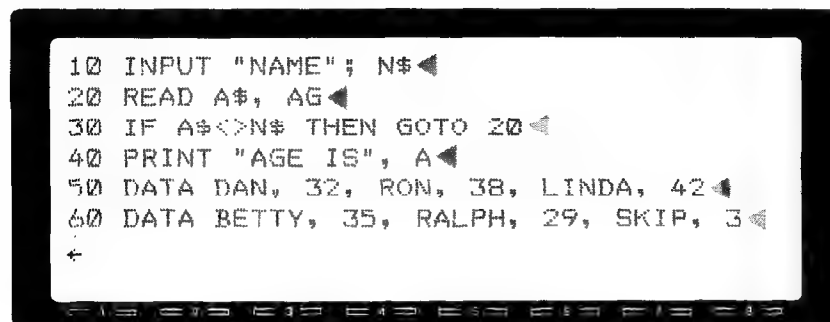
Obviously, the program is not working correctly. The correct age is 42, not 0. The problem is in line 40. Variable A at the end of this line should be changed to AG. Line 40 should appear as

```
40 PRINT "AGE IS", AG
```

This error could be corrected by retyping the line as shown above. But instead, we'll use the Editor to change the variable name to AG by inserting the single character G after A. Type the command

```
EDIT (ENTER)
```

You will see the following displayed:



```
10 INPUT "NAME"; N$◀
20 READ A$, AG◀
30 IF A$<>N$ THEN GOTO 20◀
40 PRINT "AGE IS", A◀
50 DATA DAN, 32, RON, 38, LINDA, 42◀
60 DATA BETTY, 35, RALPH, 29, SKIP, 3◀
◀
```

The Computer enters the editor mode and the program is displayed. The cursor is on the first character of the program. The symbol

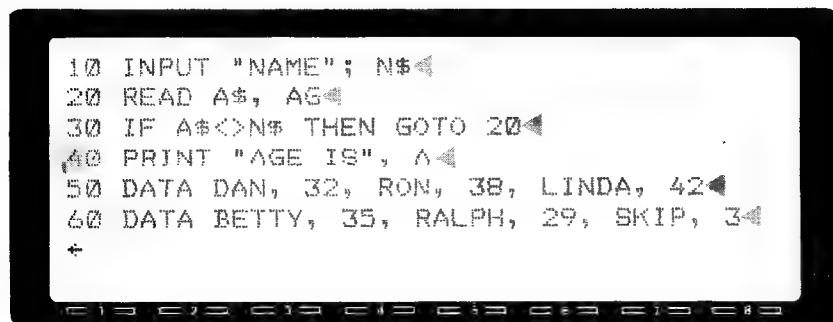


at the end of each line is used to display the carriage return character that is generated when (ENTER) is pressed. The ◀ displayed after line 60 is an **end-of-file** marker.

To insert the "G" after the variable A in line 40, the cursor must be moved so that it is directly over the ◀ in line 40.

The cursor is moved using the four Cursor Movement Keys in the upper right corner of the keyboard. The arrows indicate which way the cursor will be moved when the key is pressed.

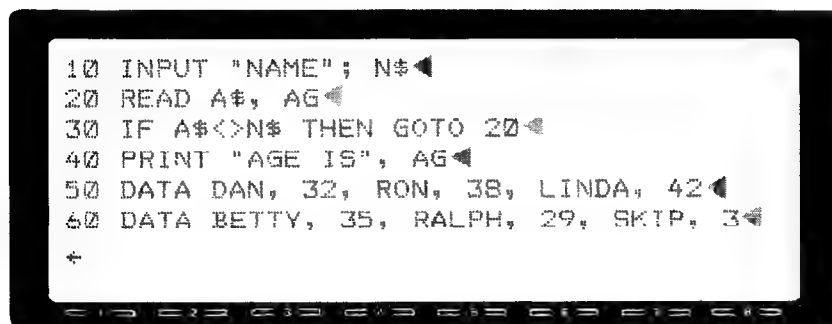
Press the down arrow key three times. This should place the cursor on the 4 in line 40. The display should appear as:



```
10 INPUT "NAME"; N$  
20 READ A$, AG  
30 IF A$<>N$ THEN GOTO 20  
40 PRINT "AGE IS", A$  
50 DATA DAN, 32, RON, 38, LINDA, 42  
60 DATA BETTY, 35, RALPH, 29, SKIP, 3  
←
```

The cursor must now be moved to the right until it is over the last character (the triangle) in line 40. If the right arrow key is pressed and held down, the cursor will move to the right until the key is released. Using the right arrow key in this fashion, move the cursor to the desired position. If you go too far, use the back arrow key to back up to the correct position.

Once the cursor is correctly positioned, you are ready to insert the letter G. This is done simply by pressing (G). Do NOT press (ENTER) after (G) is pressed. You will notice that the letter G is displayed in the correct place and the carriage return character moves one column to the right. At this point, the display should appear as:



```
10 INPUT "NAME"; N$  
20 READ A$, AG  
30 IF A$<>N$ THEN GOTO 20  
40 PRINT "AGE IS", AG G  
50 DATA DAN, 32, RON, 38, LINDA, 42  
60 DATA BETTY, 35, RALPH, 29, SKIP, 3  
←
```

When the Editor is used, the system is always in the "insert" mode. This means that if a keyboard character is typed, this character will be inserted in the line where the cursor was placed. Characters to the right of the cursor move over to make room for the inserted character.

You must remember to use the arrow keys to move about on the display, and not the space bar or (ENTER).

Now that "G" has been inserted, you must exit from the Editor before you can execute the program. To exit from the Editor:

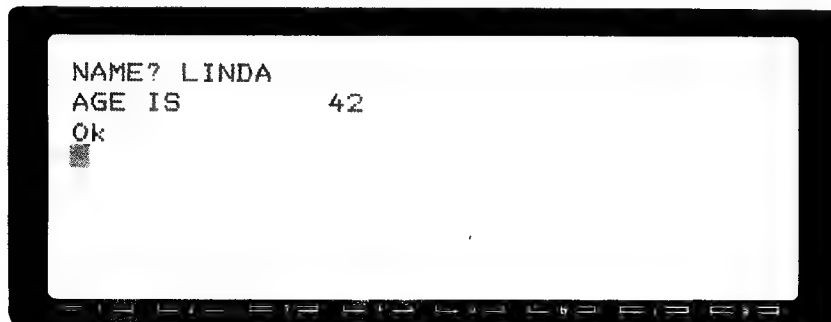
**PRESS THE ESCAPE KEY (ESC) TWICE or  
PRESS (F8) once**

---

After this is done, you will be back in BASIC with the updated program. List it to verify that it is:

```
10 INPUT "NAME"; N$
20 READ A$, AG
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "AGE IS", AG
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

Note that line 40 contains the correct variable AG. Execute the program. Here is an example of the output:



Run the program several times entering different names to verify that it is executing correctly.

## Experiment #2 Inserting a Word

Lines 10 and 40 of the previous program will be changed to

```
10 INPUT "YOUR NAME"; N$
40 PRINT "YOUR AGE IS"; AG
```

The word YOUR must be inserted in both lines as shown above. This will be done using the Editor.

Type the command

EDIT (ENTER)

to enter the Edit Mode.

The following will be displayed:

```
10 INPUT "NAME"; N$
20 READ A$, AG
30 IF A$ <> N$ THEN GOTO 20
40 PRINT "AGE IS", AG
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
←
```

The cursor is placed over the 1 in line 10. Move the cursor to the right so that the cursor is directly over the letter N in the word NAME.

Line 10 should appear as

```
10 INPUT "NAME"; N$
```

Type in the word YOUR followed by a space. Line 10 should now appear as:

```
10 INPUT "YOUR NAME"; N$
```

Line 10 is now in the desired form. Next, the cursor must be positioned on the letter A of the word AGE in line 40. This can be done using the Cursor Movement keys. Be careful not to press any other keys while you are moving the cursor. When you have the cursor positioned correctly, line 40 should appear as:

```
40 PRINT "AGE IS"; AG
```

The cursor is positioned correctly for the insertion, so type the word:

```
YOUR
```

followed by a space. Line 40 should now appear as:

```
40 PRINT "YOUR AGE IS", AG
```

At this point editing is finished, so you can exit from the Editor. Press (ESC) twice to go back to BASIC or simply press (F8).

List the program to confirm that it is:

```
10 INPUT "YOUR NAME"; N$
20 READ A$, AG
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "YOUR AGE IS", AG
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

Note that lines 10 and 40 contain the desired changes.

Execute the program. You will have to enter one of the names listed in the Data statements, or an **Out of Data** error will occur. Here is an example of the output:



```
NAME? RON
YOUR AGE IS      38
OK
█
```

Execute the program several times, entering different names.

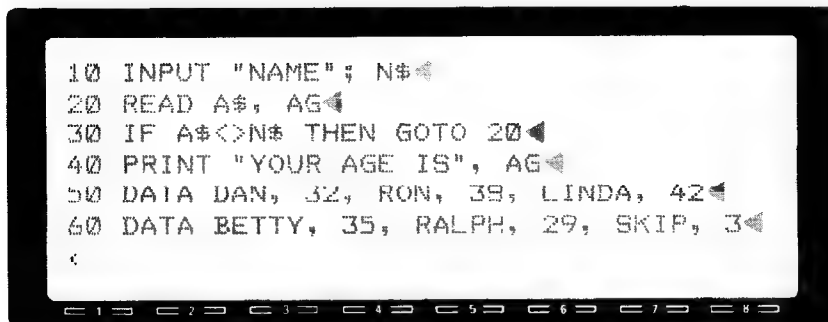
## Experiment #3 Deleting a Character

In addition to inserting a character, the Editor can be used to delete a character. This is easily done and will be illustrated in this experiment. The variable AG in lines 20 and 40 of the previous program will be changed to the single letter G.

Type the command

**EDIT (ENTER)**

to invoke the Editor. The following should be displayed:



```
10 INPUT "NAME"; N$
20 READ A$, AG
30 IF A$<>N$ THEN GOTO 20
40 PRINT "YOUR AGE IS", AG
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
<
```

The A in the variable AG in line 20 will be deleted first. Position the cursor with the arrow keys so that it is over the letter G. Line 20 should appear as

```
20 READ A$, AG
```

Note that the cursor is positioned to the right of the character to be deleted. Now press **(BKSP)**. Line 20 will now appear as

```
20 READ A$, G
```

Notice that the letter A has been deleted as desired and that the characters which were to the right of the deleted letter A have been moved to the left one position. Also note that the cursor is still positioned over the letter G.



Using the arrow keys, position the cursor over the letter G of the variable AG in line 40. When you have done this, line 40 should appear as:

```
40 PRINT "YOUR AGE IS", AG
```

Press **(BKSP)** to delete the letter A. Line 40 now should appear as

```
40 PRINT "YOUR AGE IS", G
←
```

At this point, the editing is finished. Press **(ESC)** twice to exit from the Editor. List the program to confirm it is:

```
10 INPUT "YOUR NAME"; N$
20 READ A$, G
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "YOUR AGE IS", G
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

Execute the program to confirm that it is working correctly.

It should be clear that characters can be easily deleted with the use of the Editor. You need only remember to position the cursor one character to the right of the character to be deleted.

Another very similar way to delete characters consists of entering the Editor, positioning the cursor right over the character you wish to delete and then pressing **(DEL)** (**(SHIFT)(BKSP)**).

As with **(BKSP)**, characters to the right of the deletion will shift to the left to fill the vacant space.

## Experiment #4 Changing a Character

Here is the previous program again as it currently exists in memory:

```
10 INPUT "YOUR NAME"; N$
20 READ A$, G
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "YOUR AGE IS", G
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

The comma in line 40 will be changed to a semicolon so that the age will be printed closer to the phrase "YOUR AGE IS." The comma specifies that the age will be printed in the next field. The use of a semicolon, however, will eliminate all but one space before the age. The change will be made with the Editor.

Enter the Editor with the command

```
EDIT (ENTER)
```

Again you will see displayed:

```
10 INPUT "NAME"; N$
20 READ A$, G
30 IF A$ <> N$ THEN GOTO 20
40 PRINT "YOUR AGE IS", G
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
←
```

Position the cursor so that it is on the space preceding the variable G in line 40. When this is done, line 40 should appear as

```
40 PRINT "YOUR AGE IS", G
```

Press **(BKSP)** on the keyboard. This will delete the character just to the left of the cursor. After this is done line 40 will appear as

```
40 PRINT "YOUR AGE IS" G
```

Note that the cursor is still on the space which precedes G, thus it is in the correct position to insert the semicolon. Type a semicolon. Line 40 should now appear as:

```
40 PRINT "YOUR AGE IS"; G
```

This is the desired form of line 40. You must remember that when **(BKSP)** is used, the character deleted will be the one just to the left of the cursor. If the deletion is done first, then the cursor will be positioned correctly for the insertion of the new character. The operation can be carried out in the reverse order, but after the insertion, the cursor must be moved one position to the right before the deletion is done.

Exit from the Editor by pressing the Escape key twice, or just hit **(F8)**.

Execute the program several times to confirm that it is working correctly.

## Experiment #5 Changing a Word

List the previous program. You should see:

```
10 INPUT "YOUR NAME"; N$
20 READ A$, G
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "YOUR AGE IS"; G
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
```

In this experiment, the name RALPH in line 60 will be changed to MORT, with the use of the Editor. Enter the command

EDIT (ENTER)

to activate the Editor.

Position the cursor in line 60 so that it is on the comma after the name RALPH. Line 60 should appear as:

```
60 DATA BETTY, 35, RALPH, 29, SKIP, 3
+
```

Press (BKSP) five times to delete the name RALPH. After this is done, line 60 will appear as:

```
60 DATA BETTY, 35, , 29, SKIP, 3
+
```

The cursor is correctly positioned for the insertion of the new name. Type in the name MORT. Line 60 should appear as

```
60 DATA BETTY, 35, MORT, 29, SKIP, 3
+
```

Line 60 is in its desired form; RALPH has been replaced by the name MORT. Exit from the Editor.

List the program to confirm that it is:

```
10 INPUT "YOUR NAME", N$
20 READ A$, G
30 IF A$ < > N$ THEN GOTO 20
40 PRINT "YOUR AGE IS"; G
50 DATA DAN, 32, RON, 38, LINDA, 42
60 DATA BETTY, 35, MORT, 29, SKIP, 3
```

Execute the program. Here is an example of the output:



```
NAME? MORT
YOUR AGE IS 29
OK

```

These experiments have illustrated how the Editor allows a character or characters to be inserted, deleted or changed in a program without retyping any lines. With a little practice you will become quite adept at using the Editor to make necessary changes to your program.

## Experiment #6 Changing Line Numbers

Delete the previous program from working memory with the NEW command.

The following program allows an arbitrary list of numbers to be entered from the keyboard. When a zero is entered, the average of the non-zero numbers is computed and displayed. (At least that is what the program should do.)

Type the program exactly as it is listed:

```
10 INPUT "NUMBER"; N
20 CT = CT + 1
30 AV = AV + N
40 IF N=0 THEN GOTO 60
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
```

The program accumulates the sum of the numbers in the variable AV. The variable CT is a counter that records the number of values entered. If the number entered is 0, then the average is printed in line 60.

Here is an example of the execution of the program:



Obviously, the program is not working correctly. The average of the two numbers is 3, not 2. The reason for the inaccuracy is that the number 0 was counted by the variable CT.

The program can be corrected by interchanging lines 20 and 40. In this way, the check for a zero value for N is done before the number N is counted. One way of changing the order of the statements is to retype the two lines. However, an easier method is to use the Editor to change the line numbers.

Use the command

EDIT (ENTER)

to enter the Edit Mode.

The following will be displayed:

```
10 INPUT "NUMBER"; N◀  
20 CT = CT + 1◀  
30 AV = AV + N◀  
40 IF N = 0 THEN GOTO 60◀  
50 GOTO 10◀  
60 PRINT "AVERAGE IS"; AV/CT◀  
+
```

The line number 20 must be changed to 40 and the old line number 40 must be changed to 20. Move the cursor to the 0 in line 20. When this is done, line 20 should appear as:

```
20 CT = CT + 1◀
```

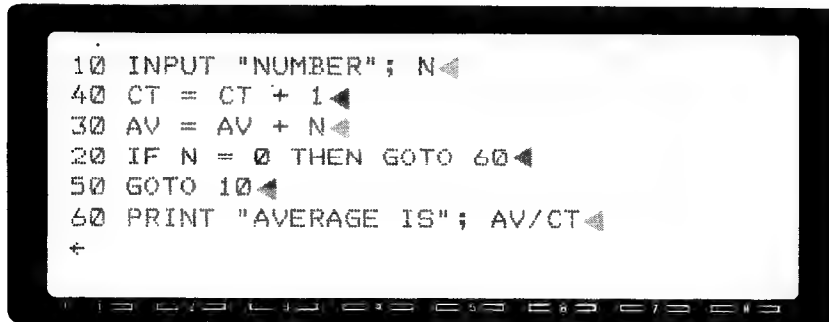
The cursor is positioned correctly to change the 2 to a 4. Delete the 2 by pressing **(BKSP)**. Then insert the 4 by pressing **(4)** on the keyboard. Now the program should appear as:

```
10 INPUT "NUMBER"; N◀  
40 CT = CT + 1◀  
30 AV = AV + N◀  
40 IF N = 0 THEN GOTO 60◀  
50 GOTO 10◀  
60 PRINT "AVERAGE IS"; AV/CT◀  
+
```

Next, move the cursor down two lines so that it is over the zero of line 40. The line should then appear as

```
40 IF N = 0 THEN GOTO 60◀
```

Delete the 4 by pressing **(BKSP)**, then insert a 2 by typing **(2)**. The program will appear as:



```
10 INPUT "NUMBER"; N
40 CT = CT + 1
30 AV = AV + N
20 IF N = 0 THEN GOTO 60
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
+
```

Although the lines are not listed in the correct order, they do have the correct line numbers.


Exit from the Editor.

List the program. You will see:

```
10 INPUT "NUMBER"; N
20 IF N=0 THEN GOTO 60
30 AV = AV + N
40 CT = CT + 1
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
```

The program is listed in the correct order, because BASIC always lists the lines in your program according to their line numbers.

Here is an example of the execution of the program:



```
NUMBER? 4
NUMBER? 2
NUMBER? 0
AVERAGE IS 2
OK
```

The program seems to be working correctly. Execute the program several times to verify that it will work correctly in every case.

You will find that the ability to move a line in your program to a different position by changing the line number is very useful and convenient.

## Experiment #7 Changing a Phrase

It is possible to move a word or even a phrase from one place in your program to another with the Editor. This can be very useful when you want to add a line, or a portion of a line to another line.

List the last program to confirm that it is:

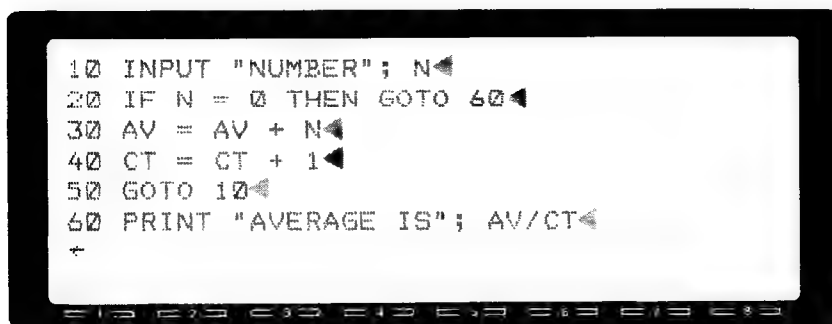
```
10 INPUT "NUMBER"; N
20 IF N=0 THEN GOTO 60
30 AV = AV + N
40 CT = CT + 1
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
```

The statement in line 40 will be placed in line 20, and line 40 eliminated. The revised program will be as follows:

```
10 INPUT "NUMBER"; N
20 IF N=0 THEN GOTO 60 ELSE CT = CT + 1
30 AV = AV + N
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
```

Enter the EDIT command.

The program will be displayed as:



```
10 INPUT "NUMBER"; N
20 IF N = 0 THEN GOTO 60
30 AV = AV + N
40 CT = CT + 1
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
+
```

First the word ELSE will be added to line 20. Position the cursor in line 20 over the carriage return character. Line 20 should appear as:

```
20 IF N = 0 THEN GOTO 60
```

The cursor is now positioned for the insertion of the word ELSE. Type a space and then the word ELSE. Line 20 should appear as

```
20 IF N = 0 THEN GOTO 60 ELSE
```

Next, the statement in line 40 must be inserted after the word ELSE in line 20. This is accomplished as follows: Position the cursor in line 40 as follows

```
40 CT = CT + 1
```

The cursor is placed on the first character (a space) to be moved. Press the **SELECT Function Key (F7)** on the top row of the keyboard. Move the cursor to the right by pressing **(→)** until it is over the carriage return character at the end of line 40. As the cursor moves, you will note that the characters are printed in **reverse video** (light on dark). This indicates which characters are going to be moved. Line 40 should appear as

```
40 CT = CT + 1
```

Press the **CUT Function Key (F6)**. When you do this, the characters marked will disappear from the screen. Line 40 should now appear as:

```
40
```

The characters to be moved to line 20

```
CT = CT + 1
```

which no longer appear on the display have been moved to a temporary storage area in the computer called the **"PASTE buffer."** The operation of deleting characters, as done above, is called a **"cut."**

The remaining characters in line 40, namely the number 40 and the carriage return, will now be deleted. Position the cursor so that it is to the right of the carriage return character.

```
40
```

Press **(BKSP)** three times. This deletes what was left of line 40. Since **(BKSP)** was used, instead of using a **"SELECT"** and **"CUT"** operation, these characters are not saved in the **"PASTE buffer."**

Your program should appear as:

```
10 INPUT "NUMBER"; N
20 IF N = 0 THEN GOTO 60 ELSE
30 AV = AV + N
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
+
```

The only remaining operation is to insert the characters saved in the PASTE buffer into line 20. Position the cursor in line 20 over the carriage return character. When this is done, line 20 will appear as

```
20 IF N = 0 THEN GOTO 60 ELSE
```



To insert the characters in the paste buffer, press the **PASTE** Command key on the keyboard. When this is done, the characters will be inserted as desired. Line 20 will now appear as:

```
20 IF N = 0 THEN GOTO 60 ELSE CT = CT +  
1←
```

*The movement of the characters from one part of the program to another requires a **SELECT**, a **CUT**, and a **PASTE**.*

After the cut operation, the characters remain in the paste buffer and can be inserted in another part of the program if desired.

Exit from the Editor. List your program to confirm that it is correct:

```
10 INPUT "NUMBER"; N  
20 IF N=0 THEN GOTO 60 ELSE CT = CT + 1  
30 AV = AV + N  
50 GOTO 10  
60 PRINT "AVERAGE IS"; AV/CT
```

Execute the program to confirm that it works as it did before.

## Experiment #8

### Copying a Phrase (without deleting it)

The previous program will be modified so that the sum of the numbers entered will be printed before the average value is printed. The revised program will be as follows:

```
10 INPUT "NUMBER"; N  
20 IF N=0 THEN GOTO 60 ELSE CT = CT + 1  
40 AV = AV + N  
50 GOTO 10  
60 PRINT "SUM IS"; AV  
70 PRINT "AVERAGE IS"; AV/CT
```

Note that the previous line 60 is now line 70 and a new line 60 has been added. These changes will be made with the Editor.

Enter the command

```
EDIT
```

The program will be displayed as usual:

```
10 INPUT "NUMBER"; N
20 IF N = 0 THEN GOTO 60 ELSE CT = CT + 1
30 AV = AV + N
40 CT = CT + 1
50 GOTO 10
60 PRINT "AVERAGE IS"; AV/CT
```

Since the new lines 60 and 70 are very similar, the easiest way to revise the program is to create a copy of line 60 and then make the necessary revisions.

Put the cursor on the 6 of line 60. Line 60 should appear as:

```
60 PRINT "AVERAGE IS"; AV/CT
```

Line 60 is to be transferred to the paste buffer. Press the SELECT Function Key (**F7**). After this is done, move the cursor down one line, so that it is on the end of file marker. Line 60 will now appear as:

```
60 PRINT "AVERAGE IS"; AV/CT
```

Note that the entire line is in reverse video, indicating that it is ready to be transferred to the PASTE buffer. This line should not be deleted when it is transferred to the buffer, so press the COPY Function Key (**F5**). Remember that **F6** transfers and deletes. Now line 60 appears as

```
60 PRINT "AVERAGE IS"; AV/CT
```

The cursor is already in the correct place for the insertion, so press the paste key. You should see:

```
60 PRINT "AVERAGE IS"; AV/CT
60 PRINT "AVERAGE IS"; AV/CT
```

The second line number 60 will now be changed to a 70. Position the cursor over the 0 in the second line number 60.

```
60 PRINT "AVERAGE IS"; AV/CT
70 PRINT "AVERAGE IS"; AV/CT
```

Press **(BKSP)** and then **(7)**, to change the 6 to a 7. These two lines will now appear as:

```
60 PRINT "AVERAGE IS"; AV/CT
70 PRINT "AVERAGE IS"; AV/CT
←
```

Line 70 is in the desired form. Move the cursor to line 60 so that it is on the space after AVERAGE. Line 60 will appear as:

```
60 PRINT "AVERAGE IS"; AV/CT
←
```

Press **(BKSP)** until the word AVERAGE is erased. Then type in the word SUM. Line 60 will be:

```
60 PRINT "SUM IS"; AV/CT
```

The last step is to delete the characters "/CT" at the end of the line. Move the cursor to the end of the line so that it is over the carriage return character.

```
60 PRINT "SUM IS"; AV/CT
```

Press **(BKSP)** three times to erase the characters. Line 60 will be in the desired form

```
60 PRINT "SUM IS"; AV
```

Exit from the Editor. List the program to confirm that it is:

```
10 INPUT "NUMBER"; N
20 IF N=0 THEN GOTO 60 ELSE CT = CT + 1
40 AV = AV + N
50 GOTO 10
60 PRINT "SUM IS"; AV
70 PRINT "AVERAGE IS"; AV/CT
```

Execute the program. Here is an example of the output:



Recall that if a BASIC program is SAVED in RAM but changes are made to it when the program is LOADED in working memory, the program SAVED in RAM will reflect those changes.

This is true also of the Editor. If the Editor is used to modify a program which was previously SAVED in RAM, then these changes also appear in the SAVED program.

You should practice using the Editor to modify your programs. You will find that it is a very convenient and quick way of making changes.

## **What you have learned:**

You have learned how to use the Editor to modify your BASIC programs. In most cases, it is easier and quicker to make changes with the Editor, than to make them by retyping entire lines.

---

## Lesson #7 Sales Trend

In this Lesson, you will learn how to create a program loop with a predetermined number of repetitions. This is a useful technique when combined with subscripted variables, which will also be introduced in this lesson.

### Experiment #1 Sales Trend

The program below is a "Sales Trend" program. Its purpose is to help predict future sales based upon the trend of previous sales. The concept underlining this program is to find a straight line which best fits the historical data and then to project this line into the future.

Clear working memory with the NEW command and enter the following program:

```
10 CLS
20 INPUT "NUMBER OF PERIODS"; N
30 FOR X = 1 TO N
40 PRINT "SALES FOR PERIOD"; X;
50 INPUT Y
60 SX = SX + X:XX = XX + X*X
70 SY = SY + Y:XY = XY + X*Y
80 NEXT X
90 B = (N*XY - SX*SY) / (N*XX - SX*SX)
100 A = (SY - B*SX) / N
110 PRINT "FORECAST FOR PERIOD X IS"
120 PRINT A;" + ";B;" * X"
```

Execute this program.

The program begins by asking you to enter the number of periods of historical sales data. Type 6 and press **(ENTER)**.

The program then prompts you for the sales data for each of the six time periods. Enter the following sales data:

```
SALES FOR PERIOD 1? 103
SALES FOR PERIOD 2? 110
SALES FOR PERIOD 3? 109
SALES FOR PERIOD 4? 120
SALES FOR PERIOD 5? 119
SALES FOR PERIOD 6? 133
```

The formula for a straight line is:

$$Y = A + B * X$$

where **Y** stands for Sales and **X** for the time period number. **A** is called the "intercept" and **B** the "slope" of the line. The program uses the sales data which

CLS
FOR/NEXT
STEP
DIM
Nested Loops
Subscripted Variables
Multiple Statements

you have just entered to compute the values of A and B. If you entered the data exactly as shown above, the program will print:

```
FORECAST FOR PERIOD X IS  
96.866666666666 + 5.3714285714286*X
```

You can now use this formula to predict any future period sales by plugging an appropriate value for X. For example, to predict period 7 sales, enter the following from the keyboard:

```
PRINT 96.8667 + 5.37143 * 7
```

The resulting number, 134.467, represents the trend line forecast of period 7 sales.

You can use the program to calculate the trend line formula for any number of time periods and any sales data. Try running the program again with your own data. You can use any convenient time period you wish, such as day, week, month, quarter, or year.

## How the Sales Trend Program Works

Look at the listing of the Sales Trend program and compare it to the flowchart in Figure 7-1.

**Line 10** The CLS statement clears the display.

**Line 20** The INPUT statement displays the prompt message "NUMBER OF PERIODS?" and then waits for data to be entered. The question mark is automatically added by the INPUT statement and should not be inserted within the quotes of the prompt message. When **(ENTER)** is pressed, the number which has been typed will be assigned to the numeric variable N.

**Line 30** The FOR statement defines the beginning of a loop which is to be repeated with successive values for the index variable X. You can think of this statement as saying:

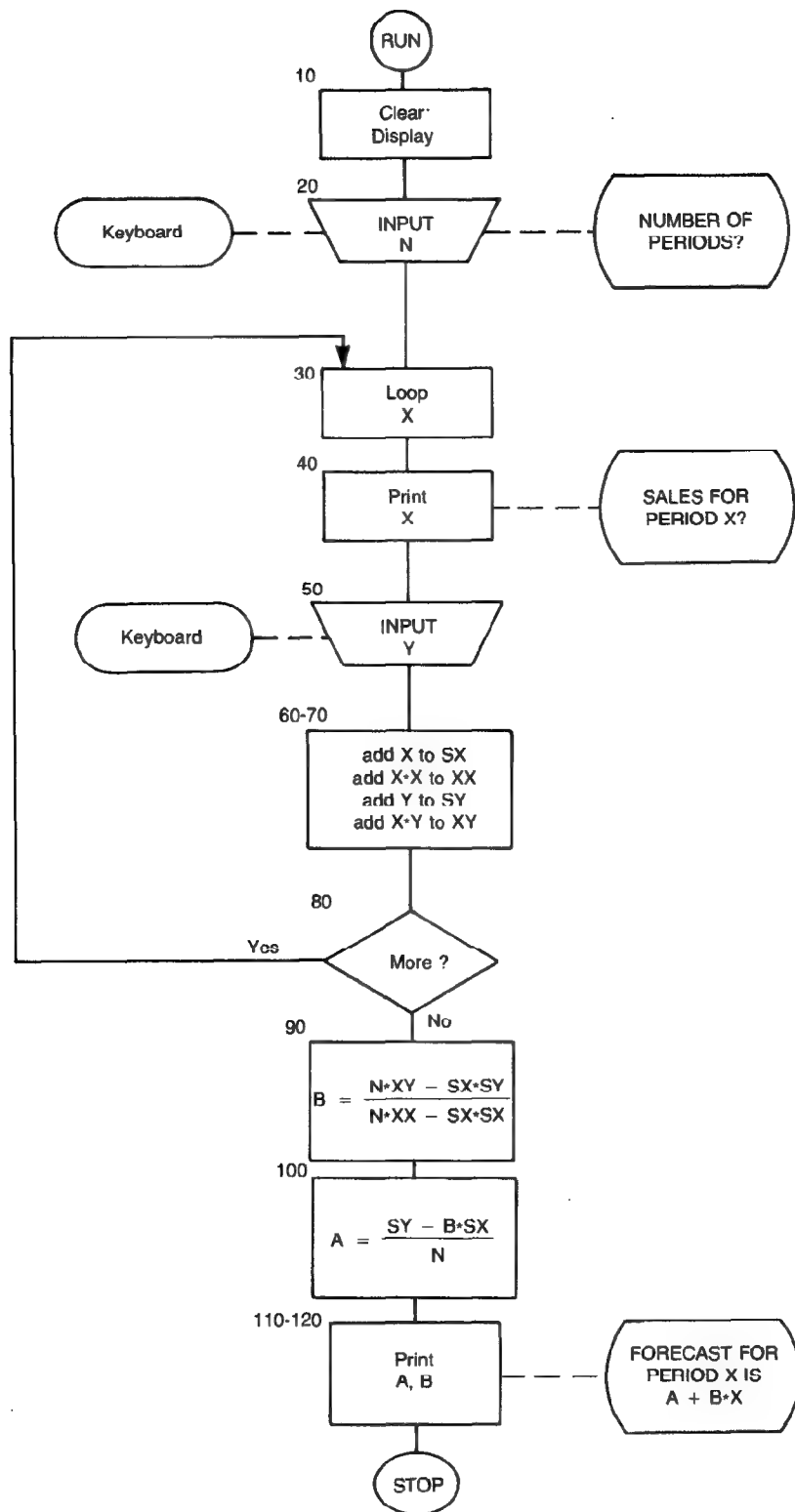
Perform the following statements with X equal to 1. Then repeat the same statements with X equal to 2. Continue repeating these same statements with successively incremented values of X (1, 2, 3, etc.). Stop repeating this loop of statements when X becomes equal to the upper limit N.

Since the variable N is INPUT during execution, this program can be used to compute a trend line for any number of sales periods.

The end of the loop is determined by a matching NEXT statement (see Line 80). When the NEXT X statement is encountered, the loop is repeated with the next value of X. If the upper limit N has been reached, execution continues with the statement following the NEXT statement.

In general, the FOR statement has an *index variable*, a *start value* and a *stop value*. The index variable must be a numeric variable. The start and stop values may be constants, variables or expressions. Another example of a FOR statement would be:

```
30 FOR A = U TO Z/(Y-2)
```



**Figure 7-1. Sales Trend Program Flowchart**

In this case, the loop of statements would be executed first with the index variable A equal to the start value stored in the variable U. The loop would then be repeated with the index variable A incremented by 1, i.e.  $A = U + 1$ . The loop would be repeated with increasing values for the variable A until the upper limit, determined by computing the value of the expression  $Z/(Y - 2)$ , is reached.

There would have to be a NEXT A statement located in the program after the FOR statement to determine the end of the loop. After the last cycle through the loop with A having the value  $Z/(Y - 2)$ , execution continues with the statement immediately after the NEXT A statement.

**Lines 40 - 50** The PRINT statement prints the prompt message

SALES FOR PERIOD X

where the value of X is determined by the previous FOR statement (Line 30). The first time through the loop, X has the value 1, so the prompt message prints as:

SALES FOR PERIOD 1

The second time through the loop, X has the value 2, so the prompt message prints as:

SALES FOR PERIOD 2

Each time through the loop, the prompt message requests the appropriate period number because the variable X is being incremented in the FOR statement.

Note the use of semicolons in this statement. The first semicolon ensures that the period number (X) will print immediately after the word "PERIOD." The second semicolon tells the Computer not to move the cursor after printing the period number. This means that the question mark (?), automatically printed by the following INPUT statement, will appear immediately after the period number.

The INPUT statement waits until a number is entered from the keyboard and stores this number in the variable Y. The variable Y acts as a temporary storage location for the current period's sales amount.

**Lines 60 - 70** These lines actually include two statements each. Line 60, for instance, includes two assignment statements separated by a colon(:). Line 60 could have been written in an equivalent manner with two separate line numbers as:

```
60 SX = SX + X
65 XX = XX + X*X
```

The two statements were put on the same line strictly as a matter of convenience and to illustrate that *Multiple Statements* (two or more statements) *can share the same line number if they are separated with a colon.*

The formula for a straight line requires the summation of several quantities:

- the variable SX is used to store the sum of the X values,
- the variable XX is used to store the sum of the squared X values, (i.e. sum of  $X*X$ )
- the variable SY is used to store the sum of the Y values, and
- the variable XY is used to store the sum of X times Y (i.e., the sum of  $X*Y$ .)

All variables in a program are initially set to zero by the RUN command. Each time through the loop, the variable X is incremented to the next period number, and Y is read in as that period's sales.



The first time through the loop, therefore, SX will be replaced with the sum of zero, the initial value stored in the variable SX, and one (the first period number). Similarly, XX will be replaced with the sum of zero and one times one. The variable XX now has the value one. The variable SY is replaced with the sum of zero and the first period's sales (103 in the example above), and the variable XY is replaced with the sum of zero and one times the first period's sales (again, 103 in the example).

Table 7-1 below illustrates how the four variables SX, XX, SY and XY sum up the appropriate quantities on each repetition of the loop:

Loop cycle	X	Y	X+X	X+Y	SX	XX	SY	XY
0	0	0	0	0	0	0	0	0
1	1	103	1	103	1	1	103	103
2	2	110	4	220	3	5	213	323
3	3	109	9	327	6	14	322	650
4	4	120	16	480	10	30	442	1130
5	5	119	25	595	15	55	561	1725
6	6	133	36	798	21	91	694	2523

**Table 7-1. Sales Trend Line Calculations**

**Line 80** The **NEXT X** statement determines the end of the loop of statements which are to be repeated with successive values for the index variable X. So long as the value of X is less than the upper limit specified in the matching **FOR X** statement, the loop will start over again with the next value for X.

When X reaches its upper limit (N in this case), execution transfers to the statement immediately following the **NEXT X** statement (line 90 in this case).

*There must be a NEXT statement to match every FOR statement in a program; without a NEXT statement, there would be no way to determine the end of the FOR loop, and therefore no way to know when to repeat the loop.*

**Lines 90 - 100** The assignment statements compute the slope, B, and intercept, A, which define the straight line which best fits the sales data. The use of parentheses was necessary in the expression to properly compute the ratios:

$$B = \frac{N * XY - SX * SY}{N * XX - SX * SX}$$

and

$$A = \frac{SY - B * SX}{N}$$

Using the sample data entered above, A and B are computed as:

$$B = \frac{6 * 2523 - 21 * 694}{6 * 91 - 21 * 21} = 5.37143$$

and

---


$$A = \frac{694 + 5.37143 * 21}{6} = 96.8667$$

**Lines 110 - 120** The PRINT statements display the resulting trend line equation:

```
FORECAST FOR PERIOD X IS
96.8666666666 + 5.3714285714286 * X
```

Line 110 prints the text contained within the quotes:

```
"FORECAST FOR PERIOD X IS"
```

Line 120 then prints the equation for the predicted sales in period X. Note the use of the semicolons to keep everything printed immediately adjacent to one another. Note also that the A and B in this statement are not enclosed in quotes, so that the values stored in the variables A and B are printed. All other items in this statement are enclosed in quotes and are therefore string constants, which print out exactly as specified within the quotes.

Since Line 120 is the last statement in the program, execution terminates after the print.

The FOR / NEXT statement pair introduced in this lesson is very useful when you need to repeat the same set of procedures a predetermined number of times. Since this situation comes up regularly in computer programming, you will no doubt want to use the FOR / NEXT pair quite often in your own programs.

The following experiments will give you a few more ideas for its application and show you that the FOR statement offers even more flexibility.

## Experiment #2 Arrays

This experiment will store the sales data as part of the Sales Trend program, rather than ask you to input it during execution.

This will make it easier to try several experiments on the data without having to retype it each time the program is run.

Change the Sales Trend program by entering the following lines:

```
20 N = 24
25 DIM Y(24)
40
50 READ Y(X)
70 SY = SY + Y(X):XY = XY + X*Y(X)
200 DATA 160, 175, 140, 230
210 DATA 155, 215, 155, 225
220 DATA 215, 265, 220, 325
230 DATA 225, 270, 265, 290
240 DATA 275, 350, 255, 345
250 DATA 300, 330, 315, 380
```

---

LIST the program; you should now have:

```
10 CLS
20 N = 24
25 DIM Y(24)
30 FOR X = 1 TO N
50 READ Y(X)
60 SX = SX + X:XX = XX + X*X
70 SY = SY + Y(X):XY = XY + X*Y(X)
80 NEXT X
90 B = (N*XY - SX*SY) / (N*XX - SX*SX)
100 A = (SY - B*SX) / N
110 PRINT "FORECAST FOR PERIOD X IS"
120 PRINT A;" + ";B;" * X"
200 DATA 160, 175, 140, 230
210 DATA 155, 215, 155, 225
220 DATA 215, 265, 220, 325
230 DATA 225, 270, 265, 290
240 DATA 275, 350, 255, 345
250 DATA 300, 330, 315, 380
```

Suppose the data in lines 200 through 250 represent quarterly sales figures for six consecutive years. RUN the program to compute the trend line on the 24 quarters of data. You will not have to enter any data during execution, since the program reads the data from the DATA statements. If you have entered the program changes and data correctly, you will see the trend line equation:

```
FORECAST FOR PERIOD X IS
148.22463768116 + 8.4086956521739 * X
```

Of course, the time period assumed here is a quarter, so that the equation will predict the sales for a specified quarter in the future. For example, to predict the first quarter of year seven (period 25), enter:

```
PRINT 148.225 + 8.4087 * 25
```

(The numbers are rounded off to three or four decimal places.) The result, 358.4425 which is displayed represents the trend line projection of sales to the next period in the future.

A new type of variable, the **subscripted variable**, has been introduced in this experiment. While this experiment could have been performed without the use of a subscripted variable, including it now will make the next experiment much easier.

The **DIM Y(24)** statement in line 25 defines the variable Y as a subscripted variable having a maximum of 24 storage locations allocated to it. These 24 storage locations can be thought of as 24 separate variables:

```
Y(1), Y(2), Y(3), . . . Y(24)
```

The number within the parentheses is called the **subscript**, and refers to the relative position of the variable within the block of storage locations set aside by the corresponding DIM (for "DIMension") statement. The block of storage locations is called the **Y array**. Thus, the variable Y(X), used in lines 50 and 70, refers to position X within the Y array.

Line 50 will read the next data item from the DATA statements and store it in position X of the Y array. Since the FOR statement repeats line 50 with X assuming values from 1 to 24, the data items will be stored in positions 1 through 24 in the Y array.

Each time through the loop, the assignment statements in line 70 will use the value stored in the next location in the Y array to compute the sums SY and XY.

Note that the Y array will contain all 24 data values when the program terminates. The next experiment will make use of this feature.

## Experiment #3 Seasonal Data

Frequently, sales data exhibit seasonal characteristics. For example, the first quarter might traditionally be slow compared to the rest of the year. If this is the case, it might be useful to modify the trend line forecast by the amount that the quarter is typically above or below the trend.

The amount above or below the trend is called the “ratio to trend” and is determined by comparing the historical (actual) sales for the quarter to the amount which the trend line would have predicted for that period.

Since there are six years of data, we compute the ratio for each of the first quarters and take the average (sum the ratios and divide by six). This is called the **average ratio to trend** and is the amount that sales in the first quarter differ, on the average, from the trend line.

Figure 7-3 illustrates the ratio to trend for quarter 1 using the example data introduced in Experiment 2 above.

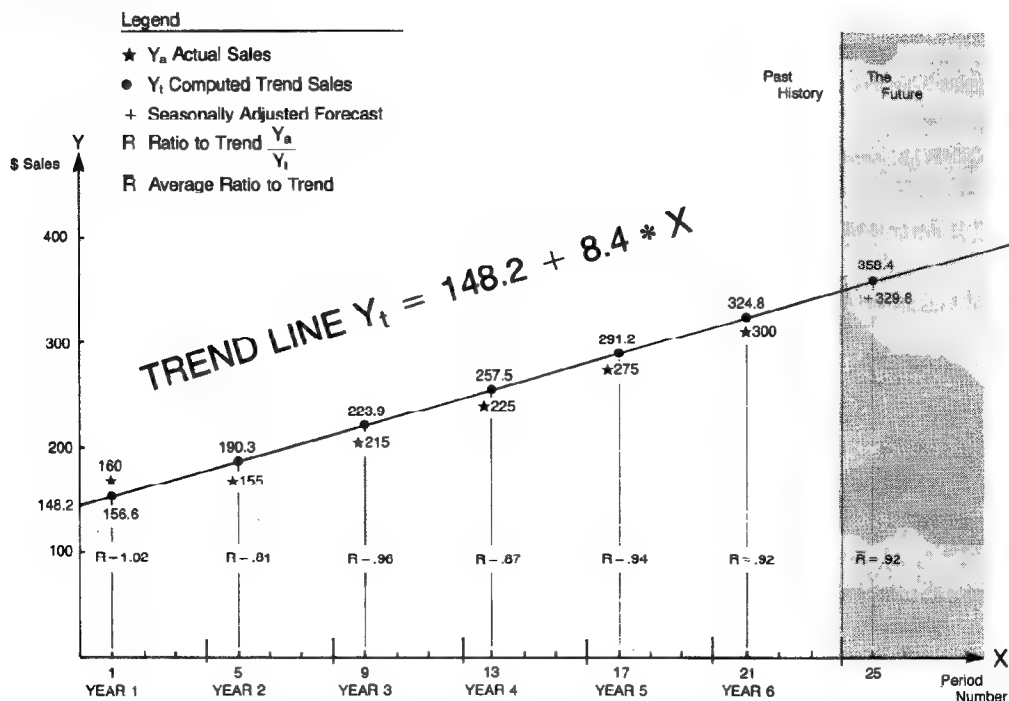


Figure 7-3. Illustration of Ratio to Trend

Table 7-2 below summarizes the average ratio to trend calculations.

Year	n Period	Y <sub>a</sub> Actual Sales	Y <sub>t</sub> 148.2 + 8.41*n Trend Value	Y <sub>a</sub> / Y <sub>t</sub> Ratio to Trend
1	1	160	156.6	1.02
2	5	155	190.3	0.81
3	9	215	223.9	0.96
4	13	225	257.5	0.87
5	17	275	291.2	0.94
6	21	300	324.8	0.92
Sum of the ratios				5.52
Average ratio to Trend				0.92

**Table 7-2. How the Average Ratio to Trend is Calculated**

To predict sales for the first quarter into the future, you would compute the trend line forecast and then multiply by the average ratio to trend for the first quarter to adjust for the season. The program can be easily modified to compute the average ratio to trend for the first period. Enter the following changes:

```

140 FOR X = 1 TO N STEP 4
150 R = Y(X) / (A + B*X) + R
160 NEXT X
170 PRINT "QUARTER 1 RATIO IS"; R/6

```

If you LIST the complete Sales Trend program, you should now have:

```

10 CLS
20 N = 24
25 DIM Y(24)
30 FOR X = 1 TO N
50 READ Y(X)
60 SX = SX + X:XX = XX + X*X
70 SY = SY + Y(X):XY = XY + X*Y(X)
80 NEXT X
90 B = (N*XY - SX*SY) / (N*XX - SX*SX)
100 A = (SY - B*SX) / N
110 PRINT "FORECAST FOR PERIOD X IS"
120 PRINT A;" + ";B;" * X"
140 FOR X = 1 TO N STEP 4
150 R = Y(X) / (A + B*X) + R
160 NEXT X
170 PRINT "QUARTER 1 RATIO IS"; R/6
200 DATA 160, 175, 140, 230
210 DATA 155, 215, 155, 225
220 DATA 215, 265, 220, 325
230 DATA 225, 270, 265, 290
240 DATA 275, 350, 255, 345
250 DATA 300, 330, 315, 380

```

RUN the new program to confirm that the trend line equation and the first quarter average ratio to trend are printed as:

```
FORECAST FOR PERIOD X IS
148.22463768116 + 8.4086956521739 *
X
QUARTER 1 RATIO IS .9230187061999
Ok
```

To predict sales into the future for quarter 1 of year 7, first compute the trend line forecast for period 25 (you can reduce the above figures to 3 or 4 decimal places):

```
PRINT 148.225 + 8.4087 * 25 (ENTER)
```

the result will be 358.4425

Then multiply by the first quarter average ratio to trend to take seasonal fluctuation into account:

```
PRINT 0.92 * 358.443
```

the answer will be 329.76756

Thus, 329.76756 would represent a prediction of quarter 1 sales in year 7, taking into account both long term trend and typical first quarter seasonal variation.

This experiment required the program to use every fourth number in the sales array Y(X). This was easily accomplished with a slight change to the FOR statement in line 140:

```
140 FOR X = 1 TO N STEP 4
```

This change in the FOR statement tells the computer to increment the index variable X in steps of 4 starting with the value 1 until X reaches the upper limit N. Thus, X will assume the values 1, 5, 9, 13, 17 and 21, which corresponds to the first quarter period number in years 1, 2, 3, 4, 5 and 6 respectively.

**Line 150** computes the ratio to trend for period X and then adds this to the sum of the previous periods ratios.

**Line 160** tests the value of the FOR statement index variable X to see if it has reached the upper limit N. If it has not, the loop consisting of lines 140 to 160 is repeated. If the index variable X has reached its upper limit, execution continues with line 170.

**Line 170** prints the average ratio to trend for quarter 1. Note that the numerical expression R/6 within the PRINT list computes the average ratio by dividing the sum of the ratios R by the number of ratios 6.

This experiment took into consideration the seasonality of quarter 1. The next experiment extends this concept to each of the four quarters.

## Experiment #4 Four Seasons

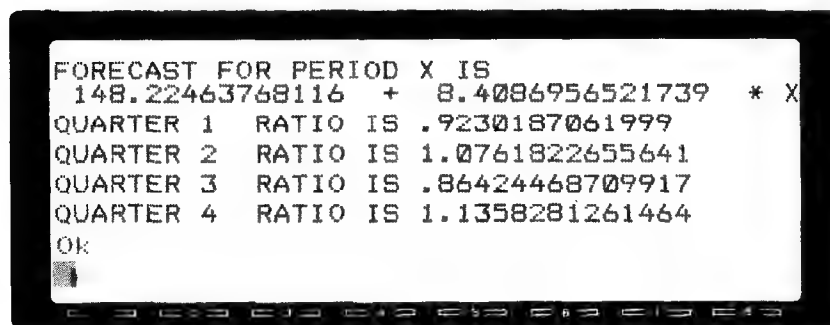
To compute the average ratio to trend for the second quarter, you could simply change the start value of X to 2 in line 140:

```
140 FOR X = 2 TO N STEP 4
```

A start value of 3 would compute the third quarter ratio and a start value of 4 would compute the fourth quarter ratio. The message "QUARTER 1 RATIO IS" would still appear for every quarter since nothing has been done to line 170. Rather than manually change line 140 for each quarter, however, you could modify the program to do this for you. Enter the following changes to the program:

```
130 FOR Q = 1 TO 4
135 R = 0
140 FOR X = Q TO N STEP 4
170 PRINT "QUARTER";Q;" RATIO IS";R/6
190 NEXT Q
```

Run this program to confirm that it now prints the average ratio to trend for all four quarters:



```
FORECAST FOR PERIOD X IS
148.22463768116 + 8.4086956521739 * X
QUARTER 1 RATIO IS .9230187061999
QUARTER 2 RATIO IS 1.0761822655641
QUARTER 3 RATIO IS .86424468709917
QUARTER 4 RATIO IS 1.1358281261464
OK
```

Lines 130 and 190 define a program loop which repeats for each of the four quarters. Notice that this loop contains within it another loop from line 140 to line 160. This is an example of a programming concept known as "nested loops." It simply means that there is a loop of statements within a loop of statements.

The inner FOR / NEXT loop must be completely contained within the outside loop, no overlap is allowed. In this experiment, the inner loop, lines 140 to 160, compute the average ratio to trend for a specified quarter Q, and the outer loop, lines 130 to 190, which repeats for each quarter with the index variable Q = 1, 2, 3 and 4.

Line 135 resets the ratio sum variable R to zero before starting the calculation for each quarter. This is now necessary to avoid starting with the sum remaining from the previous quarter.

Line 170 was changed to print the quarter Q along with the average ratio to trend R/6.

## Experiment #5 Save the Data

The sales data from the previous experiment will be used again in later lessons. To save yourself the time required to retype the data, you can simply save it as a RAM file and merge it later.

Delete lines 10 through 190 from the Sales Trend program using the CUT function of the Editor. Also, change the first digit of each of the remaining line numbers from 2 to 9.

List the remaining part of the program to confirm that it is:

```
900 DATA 160, 175, 140, 230
910 DATA 155, 215, 155, 225
920 DATA 215, 265, 220, 325
930 DATA 225, 270, 265, 290
940 DATA 275, 350, 255, 345
950 DATA 300, 330, 315, 380
```

Since this data will be merged later, it must be saved as an ASCII file. Therefore, enter the command:

```
SAVE "SALES",A
```

Go to the Menu by pressing **(F8)** and confirm that file SALES.DO is listed. This file will be used in the next Lesson.

## What you have learned:

You should now be able to use the FOR / NEXT statements in your own programs to repeat a group of statements. Using the STEP option with the FOR statement will allow you to control the increment for the index variable in the FOR / NEXT loop.

You also learned that subscripted variables facilitate manipulation of data by storing it in a block of locations called an array. You can put two or more statements on one program line using the colon (:) delimiter to save time and perhaps conserve display space. You saw how the CLS statement was used to clear the display before printing.

These new statements and concepts should prove very useful to you in writing your own BASIC programs.



---

# Lesson #8 Plot Your Data

In this Lesson you will learn how to create graphs on the Liquid Crystal Display (LCD).

PRINT@

PSET

PRESET

LINE

## Experiment #1 Graphics

The purpose of the Plot Your Data program, which will be presented shortly, is to read a list of quarterly sales figures and display them on the LCD. To do this requires the ability to display both graphics and text characters.

The LCD display on your Model 100 consists of 240 X 64 individual cells which can be used to display both graphics and text characters. The following series of simple keyboard commands will serve to illustrate this.

Clear the screen by entering the following command:

**CLS**

then turn on the cell in the center of the display with the command:

**PSET (120,32)**

Turn on the cells in each of the four corners of the display with:

**CLS (ENTER)**

**PSET (0,0) : PSET (0,63) (ENTER)**

**PSET (239,0) : PSET (239,63) (ENTER)**

The cells are quite small, so you will have to look carefully to see the illuminated "dots." You have probably figured out that the first number in the parentheses determines the horizontal (X) axis position on the display, and the second number determines the vertical (Y) axis position.

Since the corners are the extreme points of the LCD display, the range of X values is 0 (left side) to 239 (right side), and the range of Y values is 0 (top) to 63 (bottom). You should experiment a little with the PSET command by turning on various cells on the LCD display.

You can erase graphic cells in a similar way. To see this more clearly, clear the LCD and turn on a few cells in the middle of the display:

**CLS (ENTER)**

**PSET(120,32):PSET(121,33):PSET(119,33) (ENTER)**

Now turn off a cell using the PRESET command:

**PRESET(120,32) (ENTER)**

Try turning off the remaining two cells. Experiment with turning cells on and off until you feel comfortable addressing cells in any position on the display.

You can draw lines very easily on your Model 100. For example, to draw a line from the upper left corner to the lower right corner of the display, enter:

```
CLS (ENTER)
LINE (0,0) - (239,63) (ENTER)
```

Similarly,

```
LINE (0,63) - (239,0) (ENTER)
```

will draw a line from the lower left to the upper right corner. As with the PSET instruction, the first number in the parentheses is the horizontal (X) axis position and the second number is the vertical (Y) axis position. The first set of coordinates is the starting cell and the second set of coordinates is the ending cell.

To erase a line, simply add a zero after the second coordinate:

```
LINE(0,63) - (239,0),0 (ENTER)
```

A simple extension of the LINE instruction makes it easy to draw a box:

```
CLS (ENTER)
LINE (30,8) - (210,56),1,B (ENTER)
```

Graphic data, such as lines and boxes, can appear on the LCD display at the same time as text. The number 1 after the second set of coordinates says to draw the line with dark cells. The letter B at the end of the instruction says to draw a box whose opposite corners are defined by the two coordinates.

To erase a box, simply change the 1 to a 0:

```
LINE (30,8) - (210,56),0,B (ENTER)
```

It is just as easy to draw a filled in box:

```
CLS (ENTER)
LINE (210,27) - (230,37),1,BF (ENTER)
```

To erase a rectangular area on the display, change the 1 to a 0:

```
LINE (213,30) - (227,34),0,BF (ENTER)
```

## Experiment #2

### Printing Text Anywhere On The LCD

The **PRINT @** (PRINT AT) statement allows printing of text in any of 320 positions on the display. These positions correspond to eight 40 character lines as illustrated in the table below:

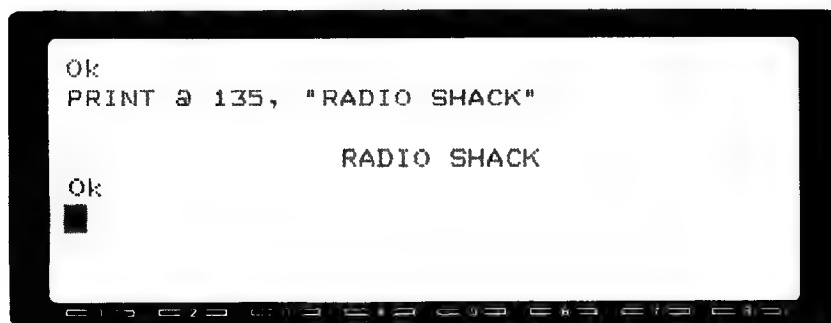
	Columns					
	1	2	3	4 . . .	. . . . 39	40
Line 1	0	1	2	3 . . . .	. . . 38	39
Line 2	40	41	42	43 . . . .	. . . 78	79
Line 3	80	81	82	83 . . . .	. . . 118	119
.						
.						
.						
Line 8	280	281	282	283 . . . .	. . . 318	319

**Table 8-1. of PRINT @ positions**

Clear the display and print the name "RADIO SHACK" in the center of the display by entering:

**CLS** **(ENTER)**  
**PRINT @ 135, "RADIO SHACK"** **(ENTER)**

You should see:



Note that the name "RADIO SHACK" begins printing in the 15th column of line 4. The PRINT@ position would be computed as:

$$40 * (4 - 1) + 15 = 135.$$

In general, to print in line L and column C, use PRINT@ position

$$40 * (L - 1) + C$$

You can print text in any order as well as any position using the PRINT@ statement, as illustrated in the following example. Enter the commands:

```
CLS:FORX=280TO35STEP-35:PRINT@X,"*";:NEXTX
```

You should see the asterisks (\*) spaced diagonally on the display as shown below:



Notice that the printing started at the bottom and proceeded to the top.

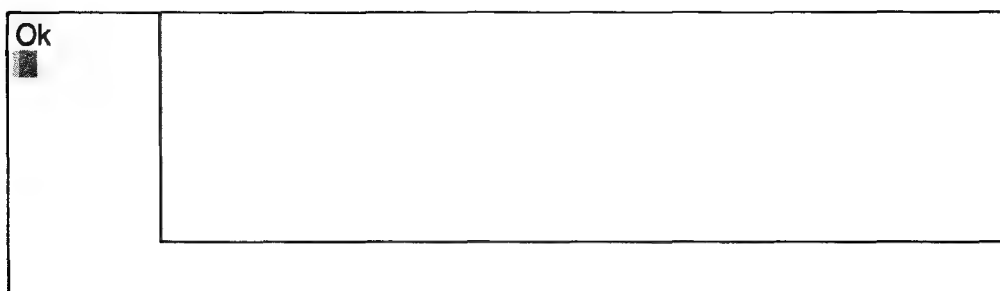
You should experiment with printing text at various locations on the display until you feel comfortable with the PRINT@ numbering scheme.

## Experiment #3 Drawing Coordinate Axes

This experiment will show you how to draw standard X-Y type axes which will allow sales data to be plotted in graph form. Enter the following program:

```
50 CLS
100 LINE(239,54)-(33,54)
150 LINE -(33,0)
```

Execute this program. You should see a pair of axes displayed:



**Line 50** This statement clears the display.

**Line 100** The horizontal axis is drawn from right to left.

**Line 150** The vertical axis is drawn from bottom to top. Note that the LINE statement contains only one coordinate,  $-(33,0)$ . This illustrates another form of the LINE statement which assumes that the first coordinate is the same as the last cell referenced in a LINE, PSET or PRESET statement. In this case, the last cell referenced was (33,54) and is used as the beginning cell for the vertical line.

---

## Experiment #4 Axis Scale

You can add "tick" marks to your axes to indicate a relative scale. Clear the display and list your program to confirm that it is:

```
50 CLS
100 LINE (239,54)-(33,54)
150 LINE -(33,0)
```

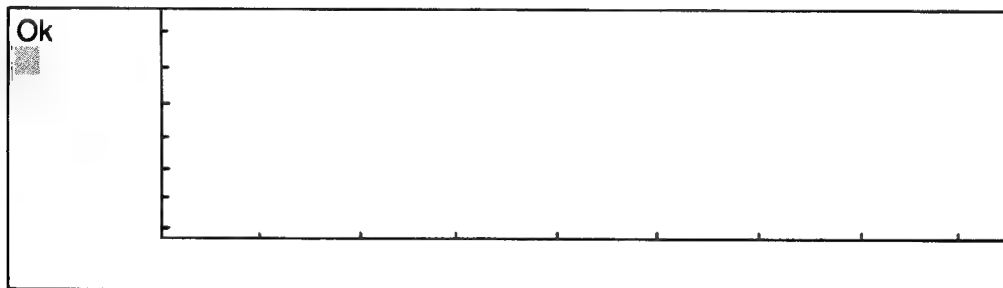
Enter the following new lines to your program:

```
200 FOR X=33 TO 239 STEP 24
300 PSET (X,53) : NEXT X
500 FOR Y=51 TO 0 STEP -8
600 PSET (34,Y) : NEXT Y
```

List the entire program to confirm that it is now:

```
50 CLS
100 LINE (239,54) - (33,54)
150 LINE -(33,0)
200 FOR X = 33 TO 239 STEP 24
300 PSET (X,53) : NEXT X
500 FOR Y = 51 TO 0 STEP -8
600 PSET (34,Y) : NEXT Y
```

Execute this program to confirm that it now displays:



**Lines 200 - 300** This FOR/NEXT loop displays the tick marks on the horizontal axis. The first tick mark is displayed at coordinate (33,53), the second at (57,53), and so on. The last tick mark will be at (225,53).

**Lines 500 - 600** This FOR/NEXT loop displays the tick marks on the vertical axis. The first tick mark is displayed at coordinate (34,51), the second at (34,43), and so on. The last tick mark will be at (34,3).

## Experiment #5 Label the Axes

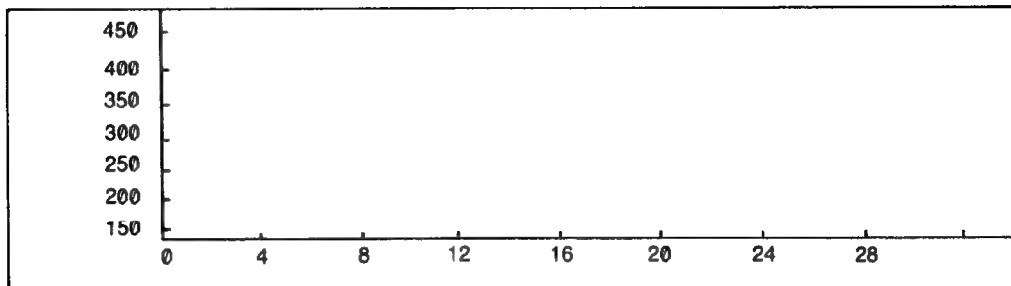
It is usually a good idea to label the scale of your graph. This experiment will show you how to print both the horizontal and vertical labels. Add the following lines to your program:

```
700 FOR X=0 TO 28 STEP 4
710 PRINT@ 284+X, X; : NEXT X
720 FOR Y=1 TO 7
730 PRINT@ 280-Y*40, 100+Y*50
740 NEXT Y
2000 GOTO 2000
```

List the program to confirm that it is now:

```
50 CLS
100 LINE (239,54) - (33,54)
150 LINE -(33,0)
200 FOR X = 33 TO 239 STEP 24
300 PSET (X,53) : NEXT X
500 FOR Y = 51 TO 0 STEP -8
600 PSET (34,Y) : NEXT Y
700 FOR X=0 TO 28 STEP 4
710 PRINT@ 284+X, X; : NEXT X
720 FOR Y=1 TO 7
730 PRINT@ 280-Y*40, 100+Y*50
740 NEXT Y
2000 GOTO 2000
```

Execute the program to confirm that it displays



**Note:** You will have to press **(BREAK)** to terminate this program.

**Lines 50 - 600** The first part of the program remains unchanged and generates the axes and the tick marks.

**Lines 700 - 710** This FOR/NEXT loop prints the labels along the horizontal axis. The horizontal (X) axis will be used to represent time in quarters for seven years, so the X variable ranges from 0 to 28. It is incremented in steps of 4 quarters, so that each year has a label. The first label (0) is printed under the origin at PRINT@ position 284, and each subsequent label is printed four columns to the right.

---

**Lines 720 - 740** This FOR/NEXT loop prints the labels along the vertical axis. The vertical (Y) axis will be used to represent dollar sales ranging from a minimum of 140 to a maximum of 380. For simplicity, labels are started at 150 and incremented in steps of 50 up to a maximum of 450. The first label (150) is printed in PRINT@ position 240 which is computed as:

$$280 - 1 * 40.$$

The second label (200) is printed in PRINT@ position 200 which is computed as:

$$280 - 2 * 40,$$

and so on.

**Line 2000** This statement creates an infinite loop. The purpose of this is to prevent the "Ok" and cursor from interfering with the display. This would occur if the program terminated execution.

## Experiment #6 Plot Sales Data

The sales data you saved in Lesson 7 will be plotted on the graph created in the previous experiment. Enter the following new statements to your program:

```
750 FOR X = 1 TO 24 : READ Y
760 PSET (33+X*6, 54-(Y-136)/6.25)
770 NEXT X
```

If you saved the Data statements under the file name SALES.DO as requested in Lesson 7, merge it with your program with the command:

**MERGE "SALES.DO"**

If you did not save file SALES.DO, simply type the data statements so that your program becomes:

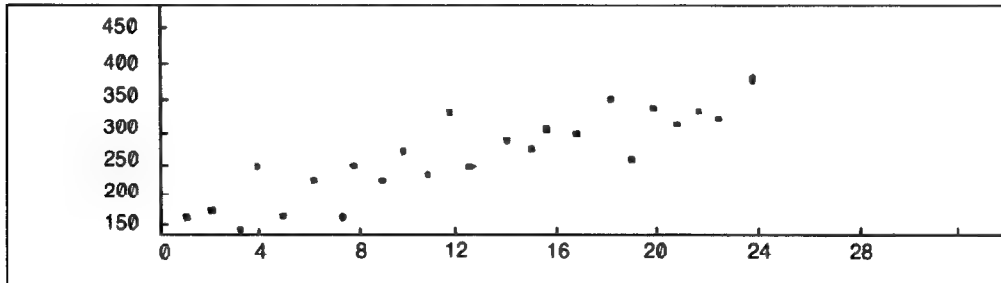
```
50 CLS
100 LINE (239,54) - (33,54)
150 LINE - (33,0)
200 FOR X = 33 TO 239 STEP 24
300 PSET (X,53) : NEXT X
500 FOR Y = 51 TO 0 STEP -8
600 PSET (34,Y) : NEXT Y
700 FOR X = 0 TO 28 STEP 4
710 PRINT@ 284+X,X; : NEXT X
720 FOR Y = 1 TO 7
730 PRINT@ 280-Y*40, 100+Y*50
740 NEXT Y
750 FOR X = 1 TO 24 : READ Y
760 PSET (33+X*6, 54-(Y-136)/6.25)
770 NEXT X
900 DATA 160, 175, 140, 230
910 DATA 155, 215, 155, 225
920 DATA 215, 265, 220, 325
930 DATA 225, 270, 265, 290
940 DATA 275, 350, 255, 345
```

---

```
950 DATA 300, 330, 315, 380
2000 GOTO 2000
```

Run this program.

You should see the following display:



You will have to press **(BREAK)** to terminate execution of this program.

This graph depicts sales as a function of time, with the horizontal (X) axis representing time in quarters of a year and the vertical (Y) axis representing sales volume.

One advantage of displaying the data in graph form is that the pattern of sales is easier to discern. In this case, for example, it is apparent that a long term upward trend in sales exists. This was not so apparent from a tabular listing of the sales data.

**Lines 50 - 740** The first part of the program remains unchanged which draws and labels the axes.

**Line 750** This begins a FOR/NEXT loop which reads the sales data from the Data statements. There are six years of four quarters, so the total number of points will be 24.

**Line 760** The PSET statement is used to turn on cells corresponding to each sales point. The axes of the graph (where  $X=0$  and  $Y=136$ ) is at graphic cell (33,54). Horizontally, the quarters are spaced six cells apart. The X coordinate of quarter one is therefore computed as:

$$33 + 1*6 = 39,$$

the second quarter as:

$$33 + 2*6 = 45,$$

and, in general, quarter X as:

$$33 + X*6.$$

The computation of the vertical coordinate is a little more complicated. One reason for this is that graphic cell vertical coordinates increase from the top to the bottom of the display, whereas the graph itself assumes that values increase from bottom to top. The horizontal axis corresponds to a value of 136, which was carefully chosen so that the labels line up with reasonable values (150, 200, etc.).



The expression:

$$Y - 136$$

computes the numerical deviation above the horizontal axis. The vertical tick marks are spaced eight graphic cells apart and represent a sales increase of 50. This means that each graphic cell is an increase of:

$$50 / 8 = 6.25$$

sales units. Thus the expression:

$$(Y - 136) / 6.25$$

is the number of graphic cells above the horizontal axis. Finally, the Y cell coordinate is measured relative to the position of the horizontal axis at vertical coordinate 54, so the expression:

$$54 - (Y - 136) / 6.25$$

is the vertical cell coordinate. For example, the first quarter sales is  $Y = 160$ . The vertical coordinate is:

$$54 - (160 - 136) / 6.25 = 54 - 3.84 = 50.16$$

which rounds to 50. The first quarter sales point is therefore graphic cell

$$(39,50).$$

**Line 770** This NEXT statement terminates the FOR loop begun in Line 750.

**Lines 900 - 950** These Data statements contain the six years of sales values in chronological order.

**Line 2000** This endless loop prevents termination of execution so that the cursor does not interfere with the graph.

## Experiment #7 Connect the Points

The readability of the graph created in the last experiment can be improved by connecting the data points with straight lines. This is rather easy on the Model 100 using the LINE statement.

Add the new line:

**745 PSET (39,50)**

and change line 760 (this is easy with the Editor) to

**760 LINE - (33 + X\*6,54 - (Y - 136)/6.25)**

List the program to confirm that it is:

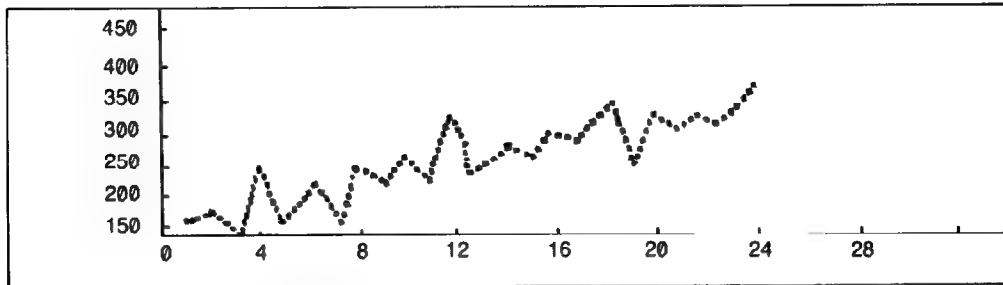
```
50  CLS
100  LINE (239,54) - (33,54)
150  LINE - (33,0)
200  FOR X = 33 TO 239 STEP 24
300  PSET (X,53) : NEXT X
500  FOR Y = 51 TO 0 STEP -8
```

```

600 PSET (34,Y) : NEXT Y
700 FOR X = 0 TO 28 STEP 4
710 PRINT@ 284+X,X; : NEXT X
720 FOR Y = 1 TO 7
730 PRINT@ 280-Y*40, 100+Y*50
740 NEXT Y
745 PSET (39,50)
750 FOR X = 1 TO 24 : READ Y
760 LINE -(33+X*6, 54-(Y-136)/6.25)
770 NEXT X
900 DATA 160, 175, 140, 230
910 DATA 155, 215, 155, 225
920 DATA 215, 265, 220, 325
930 DATA 225, 270, 265, 290
940 DATA 275, 350, 255, 345
950 DATA 300, 330, 315, 380
2000 GOTO 2000

```

Execute the program and you should see the display



You will have to press **(BREAK)** to terminate execution of this program.

**Line 745** The PSET statement turns on the graphic cell for the first quarter sales point.

**Line 760** The LINE statement draws a line from the last cell referenced to the next sales point. Recall that the first coordinate in the LINE statement is optional, and if omitted, draws a line from the last referenced cell. In this case, the second coordinate of the LINE statement becomes the first coordinate for the next line. LINE 745 is required so that the graph starts with the first data point.

## Experiment #8 Draw the Trend Line

Recall from Lesson 7 that the trend line for the sales data we have been plotting is given by the equation

$$Y = 148.225 + 8.4087 * X$$

This line may be drawn on your graph along with the data to better illustrate the long term trend of sales. The line may be drawn by specifying the two end points, that is, computing the Y value for  $X = 0$  and  $X = 28$ . The computations are performed as:

$$Y_0 = 148.225 + 8.4087 * 0 = 148.225$$

$$Y_1 = 148.225 + 8.4087 * 28 = 383.669$$

These values must then be converted to graphic cell coordinates as:

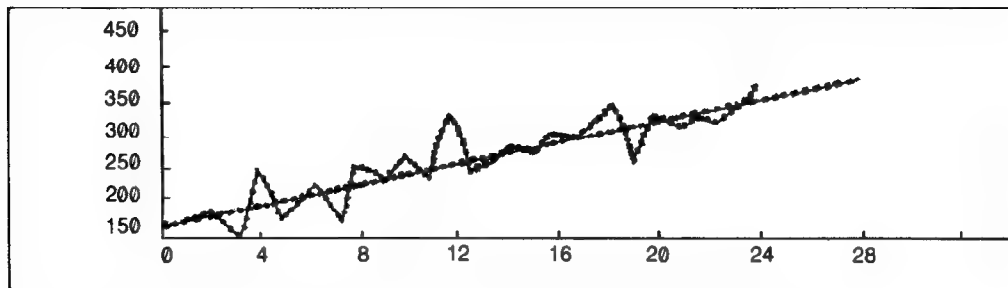
$$(33 + 0 * 6, 54 - (148 - 136) / 6.25) = (33, 52)$$

$$(33 + 28 * 6, 54 - (384 - 136) / 6.25) = (201, 14)$$

Add a new line to your program:

**800 LINE (33, 52) - (201, 14)**

and execute it to see the new display



You will have to press **(BREAK)** to terminate execution of this program.

One of the benefits of graphing the trend line is that it illustrates how next years' sales might be forecast.

## What you have learned:

In this lesson you have learned how to plot graphic data using the PSET, PRESET and LINE statements. Also you learned how to use the PRINT@ statement, which allows printing text anywhere on the display. Finally, scaling was used in assigning labels to the coordinate axes.



## Lesson #9 Functions

In this Lesson you will learn how to use certain functions to reduce the number of program lines that would be required to carry out frequently encountered tasks.

### Experiment #1 Calculating Square Root

This experiment will show you how to compute the square root of a number using the built-in function capabilities of your Computer.

To obtain the square root of a number, simply type:

**PRINT SQR(*n*) (ENTER)**

where *n* can be any positive number. For example to find the square root of 4, type:

**PRINT SQR(4) (ENTER)**

which will print the correct result:

2

Now print the square root of 2 by entering

**PRINT SQR(2)**

and observe the correct result

1.414213562373

Finally, try to print the square root of the negative number -4 by entering

**PRINT SQR(-4)**

and obtain the error message:

?FC Error

The error message indicates a "Function Call" error which occurred because the Computer cannot find the square root of negative numbers.

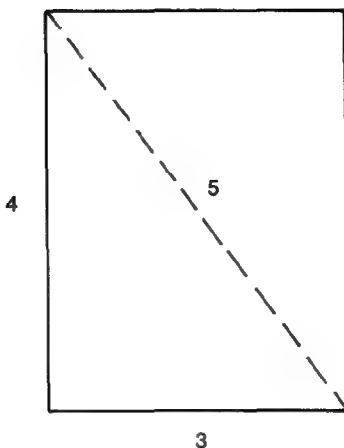
The square root function SQR(*x*) returns a numerical value for a specified numerical "argument" *x* enclosed within the parentheses. The argument must be non-negative, but can be a constant, variable or expression. For example, enter the following

**A = 3 : B = 4 : PRINT SQR(A\*A + B\*B)**

to compute the length of the diagonal of a rectangle whose sides are of length 3 and 4. The correct length is printed as

5

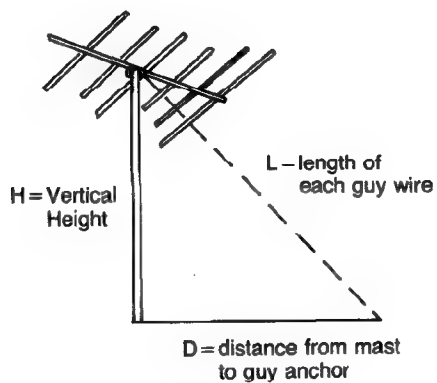
as seen in the illustration below:



## Experiment #2 Guy Wire Length

In this experiment, you will write a program to calculate the length of each guy wire required to hold up a TV mast on your roof.

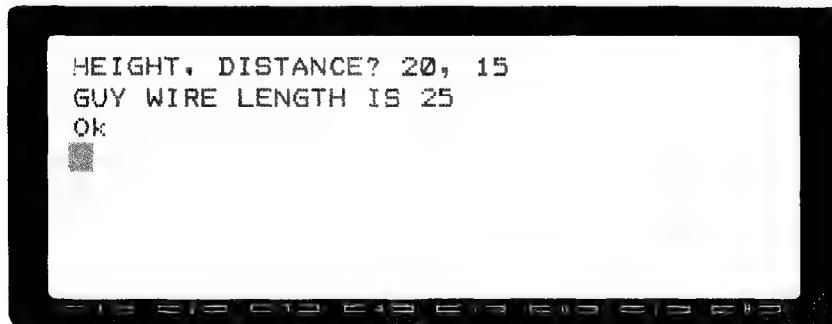
Suppose you are trying to install a TV mast on your roof and would like to precut the guy wires to the top of the pole so that you can attach them easily when the pole is stood up vertically. This is illustrated in the drawing below:



Enter the following program:

```
10 INPUT "HEIGHT, DISTANCE"; H,D
20 L = SQR(H*H + D*D)
30 PRINT "GUY WIRE LENGTH IS"; L
```

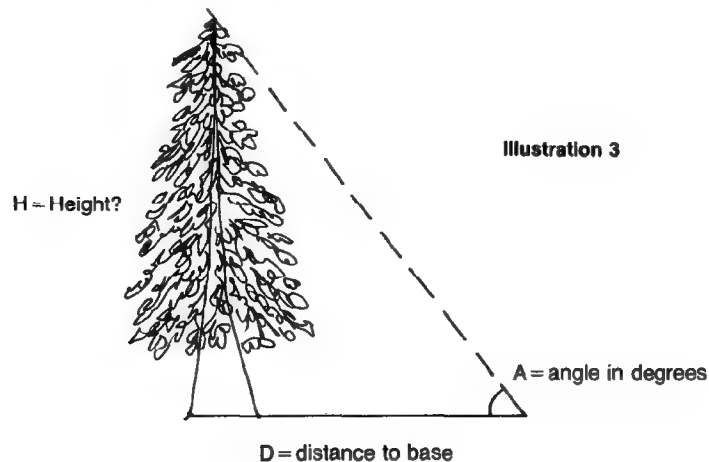
Execute this program and enter a height of 20 and a distance of 15. You should see:



Rerun the program using a height and distance of your own choosing. You might recall from geometry that this program calculates the hypotenuse of a right triangle using the Pythagorean Theorem.

### Experiment #3 Calculate the height of a tree

A problem similar to the guy wire length calculation is the determination of the height of a structure, such as a tree. The problem is illustrated below:



Using Trigonometry, the height may be found if the distance to the base and the angle to the top are known. The formula is:

$$H = D \tan \theta$$

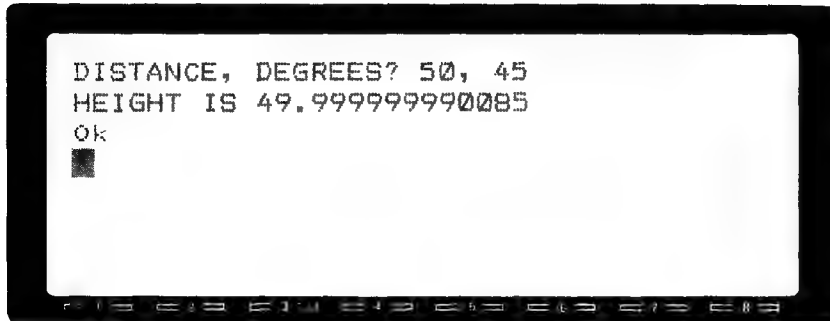
where the angle  $\theta$  (theta) is measured in radians.

Clear memory with the NEW command and enter the following program:

```
10 INPUT "DISTANCE, DEGREES"; D,A
20 PI = 4 * ATN(1)
30 R = A * PI / 180
40 H = D * TAN(R)
50 PRINT "HEIGHT IS"; H
```

Execute the program and enter a distance of 50 and an angle of 45 degrees.

The display will appear as:



Execute the program several times with different values for the distance and angle in degrees.

**Line 10** The INPUT statement allows the distance and angle in degrees to be entered from the keyboard.

**Line 20** The value of the constant PI is required to convert the angle in degrees to radians. While the constant could have been written out in decimal form, this assignment statement eliminates the need to look it up in a table or to try to remember it.

It also serves to illustrate another function which is available in BASIC, the **arctangent** (ATN). You might like to confirm that this expression calculates the constant correctly. Type:

```
PRINT 4*ATN(1) (ENTER)
```

to display the constant

```
3.1415926531932
```

**Line 30** The angle is converted from degrees to radians in this assignment statement.

**Line 40** The height is computed using the tangent function. The built-in function TAN requires the argument to be in radians.

**Line 50** The PRINT statement displays the height.



## Experiment #4 Available Memory

The Model 100 has many other useful functions built into it besides square root, tangent and arctangent. While some of these are mathematical in nature, others are more general. For example, the **FRE** function lets you determine the amount of available memory and, indirectly, the amount of memory used by your BASIC program.

To see how much memory you currently have available, type:

```
PRINT FRE(0) (ENTER)
```

The number which is displayed, such as

```
29265
```

will depend upon several factors, including how much RAM is installed in your Computer, how many files you have saved, and how large the current BASIC program is. The argument (within the parentheses) used with the **FRE** function can be a numeric constant, variable or expression. The **FRE** function will always return the amount of available memory regardless of the value of the argument. To verify this, type:

```
PRINT FRE(10) (ENTER)
```

and you should see the same value displayed as before.

To determine the amount of memory used by a BASIC program, type:

```
PRINT FRE(0) (ENTER)
```

before you begin typing your program. Then, after having typed it, type:

```
PRINT FRE(0) (ENTER)
```

again. This will print the amount of memory left or unused by your program. Finally, subtract the amount you obtained initially from the amount of memory after the program was typed. The number obtained is the number of bytes used by the program.

## Experiment #5 String Space

The **FRE** function may also be used to determine the amount of memory available to store strings. Type:

```
PRINT FRE("") (ENTER)
```

and you should see:

```
256
```

which indicates that 256 bytes of memory have been allocated for the storage of strings. The **FRE** function will return available string space if the argument is any string constant (such as the null string ""), string variable, or string expression. Verify this by typing:

```
PRINT FRE("ABC") (ENTER)
```

You can change the amount of space allocated for strings with the CLEAR statement. For example, type:

```
CLEAR 1000 : PRINT FRE("") (ENTER)
```

and you should see:

```
1000
```

which indicates that 1000 bytes have now been allocated for string space. This allocation, however, reduces the amount of available memory for your program. You can verify this by typing:

```
PRINT FRE(0) (ENTER)
```

The number displayed should be less than the previous amount available by  $(1000 - 256) = 744$  bytes.

## Experiment #6 Printing Quotation Marks

Suppose you would like to display quotation marks. This presents a problem because the quotation marks are recognized by BASIC as the delimiters of text strings. To see this, try to display the following phrase, including the quotation marks using the conventional PRINT statement:

```
PRINT "GO WEST YOUNG MAN" (ENTER)
```

The unusual result:

```
0
```

is due to the fact that the first pair of quotation marks define a string consisting of a single space.

The phrase "GO WEST YOUNG MAN" is interpreted as a numeric variable (initialized to zero). The second pair of quotation marks also print a single space after the zero. This example should make it apparent that you cannot print quotation marks in this way. However, it is possible to print them using a special string function. Type:

```
PRINT CHR$(34)"GO WEST YOUNG MAN"CHR$(34) (ENTER)
```

to display:

```
"GO WEST YOUNG MAN"
```

The function **CHR\$(34)** returns the quotation mark character as a string constant. The argument value 34 is the ASCII character code for the quotation mark. Therefore, to display a quotation mark, use **CHR\$(34)** in the PRINT statement.

Confirm this by typing:

```
PRINT CHR$(34) (ENTER)
```

## Experiment #7 Displaying ASCII Characters

Type the following program:

```
10 FOR I = 65 TO 90
20 PRINT CHR$(I);
30 NEXT I
```

and execute the program. You will see the alphabet displayed:



This program defines a loop which displays the CHR\$ function with values for the argument ranging from 65 to 90. The character returned by the CHR\$ function for these arguments is illustrated in the table below:

Argument Value	Character Returned	Argument Value	Character Returned
65	A	78	N
66	B	79	O
67	C	80	P
68	D	81	Q
69	E	82	R
70	F	83	S
71	G	84	T
72	H	85	U
73	I	86	V
74	J	87	W
75	K	88	X
76	L	89	Y
77	M	90	Z

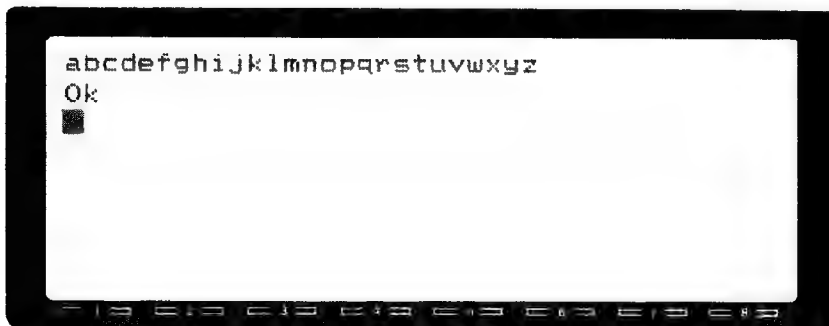
The number assigned to each letter in the table above is called its **ASCII value**. The range of possible ASCII values is 0 to 255 and includes all characters which your computer can store in its memory.

In addition to the upper case alphabet, there are ASCII values assigned to the lower case alphabet as well. These can be printed by changing line 10 in your program to:

```
10 FOR I = 97 TO 122 (ENTER)
```

---

Execute the program and it will display the alphabet in lower case:



There are quite a few other characters which can be displayed in a similar manner. Change line 10 to

```
10 FOR I = 0 TO 255 (ENTER)
```

and execute it. You should hear a beep, see the display clear and finally see several lines of characters displayed. Notice that the upper and lower case letters, the digits (0-9) and all the punctuation (comma, period, etc.) are displayed along with many other special characters (copyright symbol, graphics, and other language characters).

Not all of the ASCII values correspond to characters which can be displayed. Some of them are control codes which perform different functions, such as line feed, carriage return, sound, clear the screen and so on. For a complete list of ASCII values, refer to the Appendix in your Model 100 Owner's Manual.

Delete the current program from memory with the NEW command and type the following program:

```
10 INPUT "ASCII VALUE";A
20 PRINT CHR$(A)
30 GOTO 10
```

Execute the program and enter an ASCII value of 7 when prompted to do so. You should hear a beep. This is because the program prints CHR\$(7), and the ASCII value of 7 corresponds to the sound function. The program contains a loop back to line 10, so you will be prompted to enter another ASCII value. Enter a value of 12 and the display will clear. Enter a value of 132 and the graphics character:

⚡

for a racing car will display. Enter a value of 172 and the fraction:

1/4

will be displayed. Experiment on your own with other ASCII values. You will have to press **(BREAK)** to terminate execution of the program.

## Experiment #8 Keyboard Control of the Display

This experiment will teach you how to input keyboard characters without pressing **ENTER**. This is useful when you want the Computer to respond immediately when a key is depressed.

Clear memory with the NEW command and then enter the following program:

```
100 CLS: A = 100 : AT = A : PRINT @ A, "*"
200 A$ = INKEY$ : IF A$ = "" THEN 200
210 IF A$ = "D" THEN AT = AT + 1
220 IF A$ = "S" THEN AT = AT - 1
300 PRINT @ A, " "; : PRINT @ AT, "*";
310 A = AT : GOTO 200
```

Execute this program.

You will see an asterisk (\*) appear in the center of the display (approximately). Press **(D)**. The asterisk should move to the right. Press **(S)** and the asterisk should move to the left. You should be able to move the asterisk back and forth on the display by pressing **(D)** to move it to the right and **(S)** to move it to the left. Press **BREAK** to terminate execution of the program.

Note that this program has multiple statements on some lines to conserve display space.

**Line 100** The display is cleared and the variables A and AT are initialized to 100. The PRINT@ statement displays an asterisk in position 100, which is approximately in the center of the display.

**Line 200** The statement

```
A$ = INKEY$
```

causes the Computer to look at the keyboard to see if any key is being depressed. If a key is being depressed when the statement is executed, the INKEY\$ function will return a one character string for that key. For example, if **(D)** is being depressed when Line 200 is executed, INKEY\$ will return the one character string "D" and store it in the string variable A\$. If no key is being depressed when Line 200 is executed, INKEY\$ will return a null string (""), and A\$ will be null ("").

The second statement in Line 200:

```
IF A$ = "" THEN 200
```

causes a loop which continuously looks at the keyboard to see if a key has been depressed. If no key is depressed, A\$ is null and execution of Line 200 is repeated. When a key is depressed, A\$ will no longer be null and the condition in the IF statement will be False, causing execution to resume with the next Line, 210.

**Line 210** If **(D)** is depressed, the PRINT@ position (AT) is increased by one which will move the asterisk to the right.

**Line 220** If **(S)** is depressed, the PRINT@ position (AT) is decreased by one, which will move the asterisk to the left.

**Line 300** The statement

```
PRINT@ A, " ";
```

erases the old asterisk by printing a space over it. The second statement

```
PRINT@ AT, "*";
```

prints the asterisk at the new position. The semicolons at the end of these PRINT@ statements are required to prevent scrolling when printing in the bottom row.

**Line 310** The variable A stores the "old" asterisk position, and AT stores the "new" asterisk position. This assignment statement updates the "old" asterisk position. Execution is returned to Line 200 to allow repeated movement of the asterisk.

## Experiment #9 Move in Four Directions

Execute the previous program and hold down (S). The asterisk should move continuously to the left. When the asterisk reaches the left margin, it jumps over to the right and up one line. Continue to hold down (S) until the asterisk reaches the upper left corner of the display. If you try to move past the upper left corner, the program will terminate execution with an error message

```
?FC Error in 300
```

This "Function Call" error occurs because the PRINT@ position goes negative if (S) is pressed with the asterisk in the upper left corner (PRINT@ position 0).

The program can be modified to prevent this error from occurring. At the same time, it is relatively easy to allow the asterisk to be moved vertically. Enter the following changes into the program:

```
230 IF A$="E" THEN AT=AT-40
240 IF A$="X" THEN AT=AT+40
250 IF AT>=0 AND AT<=318 GOTO 300
260 PRINT CHR$(7); : AT=A : GOTO 200
```

List the program to confirm that it is now:

```
100 CLS : A = 100 : AT = A : PRINT @ A, "*"
200 A$ = INKEY$ : IF A$ = "" THEN 200
210 IF A$ = "D" THEN AT = AT + 1
220 IF A$ = "S" THEN AT = AT - 1
230 IF A$="E" THEN AT=AT-40
240 IF A$="X" THEN AT=AT+40
250 IF AT>=0 AND AT<=318 GOTO 300
260 PRINT CHR$(7); : AT=A : GOTO 200
300 PRINT @ A, " " : PRINT @ AT, "*";
310 A = AT : GOTO 200
```

Execute this program.

The asterisk should appear in the center of the display. Press (E) and the asterisk should move up. Press the (X) and the asterisk should move down.

As before, (S) should move the asterisk left and (D) should move the asterisk right. Hold down (E) until the asterisk reaches the top line of the display. If you attempt to move the asterisk higher, a "warning beeper" sounds and the asterisk stays on the top line.

Similarly if you try to move beyond any display boundary, the beep will sound and the asterisk will stop moving. Try this by moving the asterisk to the four corners with the appropriate keys.

**Line 230** If (E) is pressed, the PRINT@ position is decreased by 40 to move up one line.

**Line 240** If (X) is pressed, the PRINT@ position is increased by 40 to move down one line.

**Line 250** If the new PRINT@ position (AT) will be valid, that is, between 0 and 318, the condition

**AT >= 0 AND AT <= 318**

will be true, and execution jumps to Line 300. The condition uses the logical operator "AND" to combine the two logical expressions

**AT >= 0 AND AT <= 318**

into a third logical expression. A logical expression is either true or false. For example, the expression:

**AT >= 0**

is True if AT is greater than or equal to zero. It will be False if AT is less than zero. Similarly, the combined expression

**AT >= 0 AND AT <= 318**

will be True if AT is greater than or equal to zero and also less than or equal to 318. In general, if L1 and L2 are two logical expressions, then the logical expression:

**L1 AND L2**

is True if both L1 and L2 are True, and False otherwise. Logical expressions may also be combined with the OR logical operator. The expression

**L1 OR L2**

is True if either L1 or L2 is True, and False only if they are both False.

You may have wondered why the upper limit on the PRINT@ position was 318 instead of 319, which is the extreme lower right corner of the display. This was done to prevent scrolling which would occur if the asterisk was printed in the corner.

Even the use of a semicolon after the PRINT@ will not prevent the scrolling, which occurs automatically when a character is printed in position 319. You can verify this for yourself by changing Line 250 to:

**250 IF AT >= 0 AND AT <= 319 GOTO 300**

and moving the asterisk into the lower right corner.

**Line 260** This line is executed if the new PRINT@ position (AT) is outside the display limits. The statement:

```
PRINT CHR$(7);
```

causes a beep to sound. The semicolon is required to prevent scrolling off the display if the asterisk is on the bottom row.

The statement:

```
AT = A
```

sets the new PRINT@ position back to the old to keep the asterisk in the same spot on the display.

Execution is then transferred back to line 200 to continue looking at the keyboard.

## What you have learned:

In this lesson you have learned that BASIC has many useful built-in functions in addition to the mathematical operations. These include the square root, tangent and arctangent. BASIC also has general purpose functions such as FRE, CHR\$ and INKEY\$. These can be used in many other types of applications.

You also learned that BASIC allows the use of logical operators, such as AND and OR to simplify your programs. They are commonly used in IF statements.



---

## Lesson #10 Data Files

In this Lesson you will learn how to read and write data files to cassette and RAM.

Data files allow you to store information for future reference. Some common examples include:

- A list of customer names and addresses
- A list of items in inventory
- Sales data
- A list of students and their grades
- A list of hourly data readings such as temperature, pressure, humidity, wind velocity, wind direction and pollution index, etc.

You can save data files in either RAM or on cassette. The easiest and most convenient way to save a data file is in RAM. A RAM data file has the advantage of quick access and does not require attaching an external device. The disadvantage of a RAM data file, however, is that it uses up available memory space.

Cassette files are external to the Computer and do not use up valuable RAM space. When storing a large data file, it is more practical to use cassette tape. For example, a mailing list of 10000 names and addresses would require around 80K bytes of memory. This exceeds the maximum memory capacity of your Computer, so a RAM data file is out of the question. However, a list of this size could fit easily on a cassette file.

### Experiment #1 Writing a data file to RAM

The program below allows you to create a RAM file where various names may be kept.

Clear memory using the NEW command and enter the following program:

```
100 CLS
110 OPEN "RAM:NAMES" FOR OUTPUT AS 1
120 INPUT "NAME"; N$
130 IF N$ = "" GOTO 200
140 PRINT #1, N$
150 N$ = " " : GOTO 120
200 STOP
```

Execute this program and type the name John Smith when prompted, as shown below:

```
NAME? John Smith
```

The program will prompt you again to enter another name, and repeatedly do so until **(ENTER)** is pressed with no name preceding it. Enter the names as shown below:

```
NAME? Peter Wolf
NAME? Aloysius T. Cornpone
NAME? Jim Shoe
NAME? Steele Magnet
NAME? Ray D. O'Shack
NAME?
```

OPEN  
PRINT#  
CLOSE  
INPUT#  
EOF  
NOT  
MAXFILES

To terminate the program, simply press **(ENTER)** when prompted for a name.

To confirm that a data file has been created in RAM, press **(F1)** for a list of files. You should see the filename:

**NAMES.DD**

in the list of files.

Since none was specified, the extension **".DO"** was automatically added to the filename and indicating this is a **"DOcument"** file.

**Line 100** This statement clears the display.

**Line 110** The data file must be defined in an **OPEN** statement before data can be written to it. The statement:

**OPEN "RAM:NAMES" FOR OUTPUT AS 1**

defines a RAM file with the filename **"NAMES.DO"** which can be used for output with file number 1. Note that the extension to the filename will default to **".DO"** if an extension is not specified.

**Line 120** The **INPUT** statement prompts you to enter a name from the keyboard. It is stored in the string variable **N\$**.

**Line 130** The **IF** statement checks for a null entry to determine the end of the list.

**Line 140** If a file number is added to the **PRINT** statement, as in

**PRINT #1, N\$**

the items in the print list will be output to the file corresponding to the file number. The file number must have been previously defined in an **OPEN** statement. In this case, file number 1 is a RAM file.

**Line 150** The string variable **N\$** must be reset to a null string **""** in order to detect a null input in line 130. If this were not done, **N\$** would retain its last input value, and a null input could not be detected.

The program loops back to Line 120 to allow another name to be input.

**Line 200** The **CLOSE** statement terminates access to the data file and marks the end of the file.

Since this program will be used again later in this lesson, you should save the program by entering:

**SAVE "EXP1"**

## Experiment #2 Reading a RAM File

Now that you have a data file stored in RAM, you can write a program to read and display it. Clear memory using the NEW command and then enter the following program from the keyboard:

```
500 CLS:OPEN"RAM:NAMES" FOR INPUT AS 1
510 INPUT #1, N$ : PRINT N$
520 IF NOT EOF(1) GOTO 510
530 CLOSE
```

Execute this program.

You should see the display clear and then the list of names saved previously will appear as:

```
John Smith
Peter Wolf
Aloysius T. Cornpone
Jim Shoe
Steele Magnet
Ray D. O'Shack
Ok
```

**Line 500** The CLS statement clears the display. The OPEN statement defines the RAM file "NAMES.DO" which will be used for input and is referenced with file number 1. Since no extension is given in the OPEN statement, the .DO extension is assumed.

**Line 510** The INPUT #1 statement reads the next name in the data file and assigns it to the variable N\$. Note that file number 1 refers to the RAM file "NAMES.DO" as defined in the preceding OPEN statement.

The PRINT statement displays the name read from the data file.

**Line 520** The IF statement tests for the end of the data file. If it is not the end of file number 1, execution jumps back to line 510 to read another name. If the end of file number 1 is reached, execution continues with line 530. Two new features of BASIC are used in this statement.

First, the function EOF(1), returns a value of *TRUE* if the end of file number 1 has been reached, or a value of *FALSE* if the end has not been reached.

Next, the logical operator **NOT** is used to change the logical value of EOF(1).

```
IF EOF(1) is TRUE, then NOT EOF(1) will be FALSE.
IF EOF(1) is FALSE, then NOT EOF(1) will be TRUE.
```

**Line 530** The CLOSE statement terminates access to the file.

Since this program will be used later in this lesson, you should save it using the command:

```
SAVE"EXP2"
```

## Experiment #3 Saving to a Cassette File

While it is quite convenient to save data in RAM, this can use up valuable memory rather quickly. An alternative is to save the data file to cassette. Since the cassette recorder uses removable cassette tapes, you have an essentially unlimited storage capacity for your files. The disadvantage of using cassette data files is that you have to be sure the recorder is properly attached and the tape is correctly positioned for both writing and reading.

Attach your cassette recorder to the computer (consult the Owner's Manual if you have any questions). Insert a blank tape in the recorder and rewind it. Advance the tape past any leader using the fast forward key.

Load program EXP1 using the command:

```
LOAD"EXP1" (ENTER)
```

Change Line 110 to:

```
110 OPEN "CAS:NAMES" FOR OUTPUT AS 1
```

List the program to confirm that it is:

```
100 CLS
110 OPEN "CAS:NAMES" FOR OUTPUT AS 1
120 INPUT "NAME"; N$
130 IF N$ = "" GOTO 200
140 PRINT #1, N$
150 N$ = "" : GOTO 120
```

This is the same program used in Experiment 1 to write a RAM file, except that the device in the OPEN statement in LINE 110 has been changed to "CAS:".

Press the RECORD and PLAY keys on the cassette recorder together. Execute the program and enter the following names when prompted to do so:

```
NAME? Joan Smith
NAME? Patty Wolf
NAME? Allison T. Cornpone
NAME? Jill Shoe
NAME? Sally Magnet
NAME? Roxanne O'Shack
NAME?
```

Note that the cassette recorder will run for a few seconds and then stop before you are able to enter any names. Press (ENTER) without entering any name to terminate execution. After the last name is entered, the cassette recorder will run for a few seconds and then stop.

At this point, the data file "NAMES" has been written on the cassette tape. To verify this, you will have to proceed to the next experiment.

## Experiment #4 Reading a Cassette File

In this experiment, the cassette data file which you created in Experiment 3 will be read and displayed. Load program "EXP2" from RAM using the command

```
LOAD"EXP2" (ENTER)
```

and change line 500 to

```
500 CLS:OPEN"CAS:NAMES" FOR INPUT AS 1
```

List the program to verify that it is

```
500 CLS : OPEN"CAS : NAMES" FOR INPUT AS 1
510 INPUT #1, N$ : PRINT N$
520 IF NOT EOF(1) GOTO 510
530 CLOSE
```

Rewind the tape, press PLAY and execute the program. The cassette recorder will start and you will hear some sound coming from computer speaker. When the last name is read from the tape, the recorder and the sound will stop and the names will be displayed.

```
Joan Smith
Patty Wolf
Allison T. Cornpone
Jill Shoe
Sally Magnet
Roxanne O'Shack
OK
```

The only change required to change from a RAM file to a cassette file was to change the device specifier to "CAS:" in the OPEN statement in line 500.

## Experiment #5 Transfer Data from RAM to Cassette

Suppose that you have created a data file in RAM and would like to transfer it to cassette. You might like to do this to make a backup copy or perhaps to allow you to kill the RAM file to increase available memory.

The following program will allow you to transfer the file "NAMES.DO" that you created before, to cassette storage:

Clear memory with the NEW command and enter this program from the keyboard.

```
100 MAXFILES=2 : CLS
110 OPEN "RAM:NAMES" FOR INPUT AS 1
120 OPEN "CAS:NAMES" FOR OUTPUT AS 2
130 INPUT #1,N$:PRINT #2,N$:PRINT N$
140 IF NOT EOF(1) GOTO 130
150 CLOSE
```

After the program has been entered, insert a blank cassette, rewind it, and advance it past any leader. Press the PLAY and RECORD keys on the recorder and then execute the program.

The recorder will run for a few seconds and then stop; when it does, the names stored in RAM "NAMES.DO" will be displayed as:

```
John Smith
Peter Wolf
Aloysius T. Cornpone
Jim Shoe
Steele Magnet
Ray D. O'Shack
```

Finally, the recorder will run for a few more seconds as the program writes the names to the cassette file. The data file has now been transferred to cassette.

**Line 100** The MAXFILES statement limits the maximum file number which may be used, in this case, two. If you want to open more than one file at a time, you must first declare the maximum number of files with the MAXFILES statement.

As usual, the CLS statement clears the display.

**Line 110** This OPEN statement assigns the file number 1 to the RAM file NAMES.DO. Since this file will be read, it is declared an INPUT file.

**Line 120** This OPEN statement assigns the file number 2 to the cassette file NAMES. Since this file will be written to cassette, it is declared an OUTPUT file.

**Line 130** The statement INPUT #1,N\$ reads the name from the RAM file and stores it in the string variable N\$. The PRINT #2,N\$ statement writes the name to the cassette file. The statement PRINT N\$ displays the name.

**Line 140** If the end of the RAM file has not been reached, the program jumps back to line 130, where the next name will be read.

**Line 150** When the end of file is reached, both open files are closed. A CLOSE statement closes all open files. If you wish to close a specific file, add the file number to the CLOSE statement as in

```
CLOSE 1
```

You can verify that the file has been transferred properly to cassette by loading in program "EXP2," rewinding the cassette, pressing the PLAY key and running the program. If you have done everything correctly, the names will display as soon as the file has been read in.

## Experiment #6 Writing a numerical data file

In this experiment, a data file containing 30 numbers representing sales data will be created. The table below gives daily sales data for six weeks which will be saved in a RAM file for later use.

Week	Mon	Tue	Wed	Thu	Fri
1	280	275	346	280	250
2	300	260	320	300	242

```

3   292 270 350 310 255
4   310 250 310 290 260
5   280 280 290 280 270
6   285 290 330 275 258

```

Clear memory with the NEW command and enter the following program:

```

10 OPEN "RAM:SALEDA" FOR OUTPUT AS 1
20 FOR I = 1 TO 30
30 READ S : PRINT #1, S : NEXT I
40 DATA 280,275,346,280,250
50 DATA 300,260,320,300,242
60 DATA 292,270,350,310,255
70 DATA 310,250,310,290,260
80 DATA 280,280,290,280,270
90 DATA 285,290,330,275,258

```

Execute this program.

The only thing which appears to happen is that the BASIC prompt

Ok

displays after a second or two. What took place almost instantly, was that the sales data contained in the DATA statements was written to a RAM file.

List the files by pressing (F1) to confirm that file "SALEDA.DO" has been created.

**Line 10** The OPEN statement defines a RAM file with the filename "SALEDA.DO" and assigns it a file number of 1. Since the file is written to, it is declared an output file.

**Lines 20 - 30** The FOR/NEXT loop repeats 30 times, corresponding to the 30 data values. Each time through the loop, the next sales value is read from the DATA statements and then output to file number 1.

**Lines 40 - 90** The DATA statements contain the 30 sales values in chronological order.

Once the RAM file has been created, it can be used repeatedly with a variety of analysis and reporting programs. For example, you could use a statistics program to read the data file and compute the mean and median. In fact, the SALEDA.DO file will be used in the next lesson in just this way.

## What you have learned:

You should now be able to read and write data files to either RAM or cassette. Recall that RAM files are more convenient but use valuable memory. On the other hand, cassette files require an external device, but allow essentially unlimited data storage.

You also learned how the logical operator NOT may be used to simplify the condition in an IF statement.





## Lesson #11 Average Sales

In this lesson you will learn how to compute the average daily sales using the data you stored in RAM in the last Lesson under the name SALEDA.DO.

Arrays with two dimensions will be used so that the data can be examined on a daily or weekly basis. You will learn how to sort an array so that the median can be calculated. Subroutines will be used to avoid repetitive blocks of code in your programs. The TAB statement will be used to space the output neatly.

### Experiment #1 Display the File

The following program will read 6 weeks of daily sales values from file SALEDA.DO in RAM and display the values.

Clear working memory and enter the following program:

```

5  OPEN "RAM:SALEDA.DO" FOR INPUT AS 1
10  FOR W = 1 TO 6 : FOR D = 1 TO 5
20  INPUT # 1, S(W,D)
30  NEXT D : NEXT W
40  PRINT "WEEK  MON  TUE  WED  THU  FRI"
50  FOR W = 1 TO 6 : PRINT W;
60  FOR D=1 TO 5
70  PRINT TAB(D*5) S(W,D);
80  NEXT D : PRINT
90  NEXT W

```

Execute this program.

The program will read the file SALEDA.DO containing sales data.

The following table will then be output to the LCD:

WEEK	MON	TUE	WED	THU	FRI
1	280	275	346	280	250
2	300	260	320	300	242
3	292	270	350	310	255
4	310	250	310	290	260
5	280	280	290	280	270
6	285	290	330	275	258

**Line 5** The OPEN statement opens the RAM data file "SALEDA.DO" for input and assigns the file number 1 to it.

**Line 10** The first FOR statement,

```
FOR W = 1 TO 6
```

sets up an outer loop to index through the six weeks. The second FOR statement

```
FOR D = 1 TO 5
```

sets up an inner loop to index through the five days of the week.

**Line 20** The daily sales are read from the RAM file SALEDA.DO and stored in the array S. The array S, which will contain the sales values, is a two dimensional array and can be thought of as a table rather than a list. This array will have six rows and five columns. The rows correspond to the weeks and the columns the days. This may be illustrated as follows

	day				
week	MON	TUE	WED	THU	FRI
1	S(1,1)	S(1,2)	S(1,3)	S(1,4)	S(1,5)
2	S(2,1)	S(2,2)	S(2,3)	S(2,4)	S(2,5)
3	S(3,1)	S(3,2)	S(3,3)	S(3,4)	S(3,5)
4	S(4,1)	S(4,2)	S(4,3)	S(4,4)	S(4,5)
5	S(5,1)	S(5,2)	S(5,3)	S(5,4)	S(5,5)
6	S(6,1)	S(6,2)	S(6,3)	S(6,4)	S(6,5)

The sales data will be read into the array S so that the first row will contain the five sales values for the first week, the second row the sales values for the second week, etc.

**Line 30** The first NEXT statement defines the end of the inner loop and the second NEXT statement defines the end of the outer loop.

**Line 40** This line prints the "heading" for the output.

**Line 50** The first statement in the line begins another FOR / NEXT loop. This loop will display the daily sales for each week, starting with week 1, then week 2, etc., and ending with week 6. The sales were stored in chronological order, so that the first five values are the sales for the first week, the next five values are for the second week, and so on.

The second statement

```
PRINT W;
```

displays the week number. The first time through the loop, a 1 is displayed, the second time a 2, and the last time a 6 is displayed. Since a semicolon (;) follows the variable W in the print statement, the carriage return is suppressed.

**Line 60** The FOR statement defines an inner loop which increments through all five days for each week. This loop will display the five daily sales for each week.

**Line 70** The daily sales for each week are displayed on one line. The TAB statement is used to neatly space the sales values along the line. The general form of the TAB statement is

```
TAB(x)
```

and specifies that printing is to begin in column  $x + 1$ .  $x$  may be a numeric constant, variable or expression. The values of D, the expression

```
D * 5
```

and the corresponding print positions are given below:

Day D	value of D *5	print position is column
1	5	6
2	10	11
3	15	16
4	20	21
5	25	26

Examine the output closely by adjusting the Display Control dial until the column lines become clearly visible. Notice that the first digits of the sales are actually printed one column to the right of the print position specified by the TAB statement. When a number is printed, the first position is reserved for a sign. If the number is positive, then the plus sign ( + ) is not printed, but the space is still printed. If the number is negative, then the minus sign ( - ) is displayed.

Note that the variable S(W,D) is followed by a semicolon. This suppresses the carriage return so that the next sales amount printed for the week will be in the same line. Since the carriage return was suppressed in the print statement in line 50, the week number and daily sales for that week are all displayed on the same line.

**Line 80** The NEXT statement defines the end of inner loop which displays the sales for each day.

The PRINT statement in this line generates a carriage return, which causes the next week's sales to be displayed on the next line.

**Line 90** The NEXT W statement defines the end of the outer loop which causes the sales for each week to be displayed.

## Experiment #2 Compute and Display the Weekly Average

The previous program will be changed so that the weekly average sales can be computed and displayed along with the daily sales.

Change line 40 to:

```
40 PRINT "WEEK MON TUE WED THU FRI AVG"
```

and line 80

```
80 NEXT D : PRINT WA / 5
```

Also enter two new lines:

```
55 WA = 0
65 WA = WA + S(W,D)
```

List the program to confirm that it is:

```
5 OPEN "RAM:SALEDA.00" FOR INPUT AS 1
10 FOR W = 1 TO 6 : FOR D = 1 TO 5
20 INPUT * 1, S(W,D)
30 NEXT D : NEXT W
40 PRINT "WEEK MON TUE WED THU FRI AVG"
```

```

50 FOR W = 1 TO 6 : PRINT W;
55 WA = 0
60 FOR O = 1 TO 5 :
65 WA = WA + S(W,O)
70 PRINT TAB(O * 5) S(W,O);
80 NEXT O : PRINT WA / 5
90 NEXT W

```

Execute the program.

Here is what the output should look like:

WEEK	MON	TUE	WED	THU	FRI	AVG
1	280	275	346	280	250	286.2
2	300	260	320	300	242	284.4
3	292	270	350	310	255	295.4
4	310	250	310	290	260	284
5	280	280	290	280	270	280
6	285	290	330	275	258	287.6

An additional column containing the weekly averages has been printed.

The average (or mean) was calculated by adding up the five values for the week and dividing the sum by five. Line 65 calculates the weekly sum and stores it in the numeric variable WA. The variable WA is initialized to zero outside the loop (lines 60 – 80) where the sum is computed. The weekly sum is divided by five to obtain the average and displayed in line 80.

## Experiment #3 Computing the Average for Each Weekday

The previous program will be changed so that the average sales for each weekday can be calculated. There are six sales figures for each day, so to calculate the average, these six values must be added and the sum divided by six. This must be done for each of the five days.

Add the following lines to the program:

```

95 PRINT "AVG";
100 FOR O=1 TO 5 : OA=0 : FOR W=1 TO 6
110 OA = OA + S(W,O) : NEXT W
120 PRINT TAB(O*5) INT(OA/6);
130 NEXT O

```

List the program to confirm it is:

```

5 OPEN "RAM:SALEDA.DD" FOR INPUT AS 1
10 FOR W = 1 TO 6 : FOR O = 1 TO 5
20 INPUT # 1, S(W,O)
30 NEXT O : NEXT W
40 PRINT"WEEK MON TUE WED THU FRI AVG"
50 FOR W = 1 TO 6 : PRINT W;
55 WA = 0
60 FOR O = 1 TO 5

```

```

65  WA = WA + S(W,D)
70  PRINT TAB(D*5) S(W,D);
80  NEXT D : PRINT WA/5
90  NEXT W
95  PRINT "AVG";
100 FOR D=1 TO 5 : DA=0 : FOR W=1 TO 6
110 DA = DA + S(W,D) : NEXT W
120 PRINT TAB(D*5) INT(DA/6);
130 NEXT D

```

Execute this program. The output should appear as

WEEK	MON	TUE	WED	THU	FRI	AVG
1	280	275	346	280	250	286.2
2	300	260	320	300	242	284.4
3	292	270	350	310	255	295.4
4	310	250	310	290	260	284
5	280	280	290	280	270	280
6	285	290	330	275	258	287.6
AVG	291	270	324	289	255	

**Lines 5 - 90** The first part of the program remains unchanged.

**Line 95** The row label "AVG" is displayed. The semicolon (;) suppresses the carriage return so the averages will be displayed on the same line.

**Line 100** The first FOR statement

```
FOR D=1 TO 5
```

defines an outer loop which increments through each day. The assignment statement

```
DA=0
```

initializes the daily sum to zero. The second FOR statement

```
FOR W=1 TO 6
```

defines an inner loop which sums the six week's sales for a given day.

**Line 110** The sales for a given day are added and stored in the numeric variable DA. The NEXT statement defines the end of the inner loop.

**Line 120** The daily average is displayed. The TAB function is used to neatly space the output. The INT function is used to drop any decimal part for neater appearance of the output. The semicolon (;) suppresses the carriage return so that the next average will be printed on the same line.

**Line 130** The outer loop which increments through each day of the week is terminated.

By using a two dimensional array S for the sales values, it was very easy to process the data either weekly (row by row) or daily (column by column). The program could have been written using a singly dimensioned array and FOR / NEXT loops with the STEP options. However, it was easier to write the program using the two dimension array.

## Experiment #4 Sorting

The thirty sales values in file SALEDA.DO are to be sorted and printed out in ascending order. Sorting can be easily accomplished if the sales are stored in a one dimensional array.

The sorting routine which will be used is called a "bubble sort." This is a simple algorithm that is easy to understand and to program.

The bubble sort works as follows:

Starting at the beginning of the array, the first two values are compared. If the first is larger than the second, they are interchanged; otherwise nothing is done.

The second and third values are then compared. If the second is larger than the third, they are interchanged, and so forth through the entire array. When the end of the array is reached, the largest value will be stored there. The process then begins over at the beginning of the array. This time, however, it will only be necessary to go up through the next to the last element. In this fashion, the largest element goes to the end, the next largest goes to the next to the end position, etc.

The name "bubble sort" is descriptive of this process because of the way the large values pop up, one at a time, into their correct positions at the end of the array.

The following program will read in the thirty sales values, sort them and display them in ascending order. Delete the previous program from memory with the NEW command and enter the following program:

```
5   OPEN "RAM:SALEDA.DO" FOR INPUT AS 1
10  DIM S(30)
20  N = 30
30  FOR I = 1 TO 30
40  INPUT #1, S(I)
50  NEXT I
500 FOR I = 1 TO N - 1
510 FOR J = 1 TO N - I
520 IF S(J) < S(J + 1) GOTO 540
530 S=S(J) : S(J)=S(J+1) : S(J+1)=S
540 NEXT J : NEXT I
600 PRINT TAB(10) "SORTED SALES"
610 FOR R = 1 TO 6
620 FOR C = 1 TO 5
630 I = (R - 1) * 5 + C
640 PRINT TAB((C - 1) * 7) S(I);
650 NEXT C : PRINT
660 NEXT R
```

Execute the program.

There will be a pause while the program sorts the data. Then the output should appear as follows:

	SORTED VALUES			
242	250	250	255	258
260	260	270	270	275
275	280	280	280	280
280	285	290	290	290
292	300	300	310	310
310	320	330	346	350

Reading the table row by row, the values are printed in ascending order, the smallest value is 242, which is listed first and the largest value is 350 which is listed last.

**Line 5** The OPEN statement again allows input from RAM.

**Line 10** The one dimensional array S is dimensioned in this line. Remember that there are thirty sales values.

**Line 20** The numeric variable N is given the value 30. It will be convenient to use N instead of 30 in the program. If the number of sales values read in from the file is changed, only this line will have to be changed.

**Lines 30 - 50** These lines read in the data and store the values sequentially in the array S.

**Line 500** This line starts the "bubble sort" portion of the program. It is necessary to go through the array 29 times to get all the values in their correct positions. The first time through the loop ( $I = 1$ ) the largest sales value will be placed in  $S(30)$ , the second time through the loop ( $I = 2$ ), the next largest will be placed in  $S(29)$ , etc. The last time through the loop ( $I = 29$ ) the next to the smallest value will be placed in  $S(2)$ . At that point  $S(1)$  must contain the smallest value, so the array is sorted.

**Line 510** This line starts the FOR / NEXT loop in which the comparisons and possible interchanges will be made. When  $I = 1$ , J will range from 1 to  $N - 1$ , when  $I = 2$ , J will range from 1 to  $N - 2$ , and so on. When  $I = 29$ , J will only have the value 1. Thus as more and more values are stored in their correct positions at the end of the array S, fewer and fewer comparisons and interchanges need to be made.

**Line 520** This line does the comparison. If the values stored in adjacent locations,  $S(J)$  and  $S(J + 1)$ , are already in order, then execution jumps to the bottom of the inside loop. However, if they are out of order, then line 530 is executed next.

**Line 530** These three statements swap the values stored in  $S(J)$  and  $S(J + 1)$ . Note that the value stored in  $S(J)$  is stored temporarily in the variable S. This is necessary when swapping two adjacent elements in an array, to prevent erasing one of them. It is permissible to have an array S and an ordinary numeric variable S as well.

**Line 540** The two NEXT statements terminate the FOR / NEXT loops. The NEXT J statement terminates the inner loop and the NEXT I statement terminates the outer loop.

**Line 600** When the sorting is finished, this line displays a heading for the table.

**Line 610** The sorted values are displayed in six rows so that they will all fit on the display at one time. The FOR / NEXT loop which begins in this line increments once for each of the six rows.

---

**Line 620** The loop which begins in this line displays the five values in row R of the table.

**Line 630** The value of the subscript I in the S array is computed for the element displayed in row R and column C. For example, if  $R = 3$  and  $C = 2$ , then  $I = 12$ . This is necessary because the data is stored in a one dimensional array but displayed in a two dimensional table.

**Line 640** The value in row R and column C is displayed. The TAB statement is used to space the output. Note that the semicolon after S(I) suppresses the carriage return.

**Line 650** The first statement in this line terminates the inner loop which displayed the entries in row R. The PRINT statement generates a carriage return, so that the next five values will be displayed on the next line.

**Line 660** The outer loop, which displays each of the six rows, is terminated.

Sorting data is frequently required in computer programming. The bubble sort technique introduced in this experiment is a straightforward approach to this common problem, and is well worth learning. Once the sales data has been sorted, it is quite easy to compute another measure of central tendency — the median.

## Experiment #5 Computing the Median

Now that you know how to sort the sales data, you can easily change your program to compute the median. The median is similar to the average in that they both measure central tendency. The median is a number such that half of the data values are larger than the median, and half of the data values are less than the median. If the data values are sorted, the median is defined as follows

- i) if there are an odd number of values, the median is the middle value
- ii) if there are an even number of values, the median is the average of the two middle values.

In the sales data example, there are 30 data values, (an even number) and the two middle values are the 15th and 16th values. The median is the average of these two values.

Delete lines 600 through 660 and add the following two lines to the program:

```
560 MD = (S(15) + S(16)) / 2
570 PRINT "MEDIAN SALES"; MD
```

Execute the program. The values will be sorted as before, but not displayed. The following will be displayed

```
MEDIAN SALES 280
```

The median is another type of "average" which in many cases is a better measure of central tendency than the mean. For example, the median is less affected by extreme values than the mean. The median is the central value in the sense that there are just as many values above it as below it.



## Experiment #6 Computing the Median of the First N Values

The previous program will be changed so that only the first N values will be read in from the RAM file, instead of all 30. The median of these N values will then be computed and displayed.

You will be able to input the value for N when the program is executed. The main purpose of this experiment is to generalize the median calculation, and show you how to determine whether there are an even or an odd number of values.

Make the following changes to the program:

```
20 INPUT "NUMBER OF DAYS (2 - 30)"; N
30 FOR I = 1 TO N
550 N1=INT((N+1)/2) : N2=INT((N+2)/2)
560 MD = (S(N1) + S(N2)) / 2
570 PRINT "MEDIAN IS"; MD
```

List the program to verify that it is:

```
5 OPEN "RAM:SALEDA.00" FOR INPUT AS 1
10 DIM S(30)
20 INPUT "NUMBER OF DAYS (2 - 30)"; N
30 FOR I = 1 TO 30
40 INPUT # 1, S(I)
50 NEXT I
500 FOR I = 1 TO N - 1
510 FOR J = 1 TO N - I
520 IF S(J) < S(J + 1) GOTO 540
530 S=S(J) : S(J)=S(J+1) : S(J+1)=S
540 NEXT J : NEXT I
550 N1=INT((N+1)/2) : N2=INT((N+2)/2)
560 MD = (S(N1) + S(N2)) / 2
570 PRINT "MEDIAN IS"; MD
```

Execute the program.

This time you will be prompted for the number of days of sales to be read. After you enter a value, the program will read the first N values from the RAM file and display the median.

Here is an example of the execution and output of the program:

```
NUMBER OF DAYS (2 - 30)? 20
MEDIAN IS 285
```

Thus the median of the first 20 sales is 285.

Execute the program several times. Try entering both even and odd values for N. You will find that the program will correctly calculate the median in every case.

**Line 20** An INPUT statement has been added which prompts you to enter the value for N.

**Line 550** The numeric function INT is used in this line. This function drops any fractional part of a number and thus returns the greatest integer less than or equal to the argument. Consider the following examples

Argument x	Value of INT(x)
1.234	1
239.899	239
4	4

The values that are computed and assigned to the variables N1 and N2 are the middle values for the subscripts of the array S(1) through S(N).

For example, if N has the value 16, then

$$(N + 1) / 2 = 8.5$$

$$(N + 2) / 2 = 9$$

and

$$\text{INT}((N + 1) / 2) = 8$$

$$\text{INT}((N + 2) / 2) = 9$$

so that N1 and N2 will correctly contain the middle subscript values. Additional examples are given in the table below:

N		N1	N2
30	(even)	15	16
20	(even)	10	11
13	(odd)	7	7
27	(odd)	14	14

Note that when N is even, N1 and N2 are the two middle values and when N is odd, N1 and N2 are both equal to the single middle value. Thus, the use of the INT function avoids any testing to determine whether N is even or odd.

**Line 560** The median is calculated in this line. If N is even, then the average of the two middle values is computed. If N is odd, then the values stored in N1 and N2 are the same and so the median is correctly calculated as the middle value.

**Line 570** The median is displayed.

Now that you have a program which will sort an array of arbitrary length and compute the median, you can use it as a subprogram in larger programs. The next experiment shows you how to do this.

## Experiment #7 Calculating the Median for Each Weekday

In this experiment, the median sales will be calculated for each day of the week. This means that the program should compute the median value of the six Monday sales, the median value of the six Tuesday sales, and so on for each day of the week.

To accomplish this, the six values for each day must be sorted before the median can be computed. The following program illustrates an efficient method of doing this.

---

Modify the program from experiment 6 by changing lines 10 through 50 to:

```
10 N = 6
20 FOR W=1 TO 6 : FOR O=1 TO 5
30 INPUT #1, R(W,O)
40 NEXT O : NEXT W
50 FOR O=1 TO 5 : FOR W=1 TO 6
```

and enter the new lines 60 through 150

```
60 S(W) = R(W,O)
70 NEXT W
80 GOSUB 500
90 M(O) = MO
100 NEXT O
110 PRINT TAB(10) "MEDIAN SALES"
120 PRINT " MON     TUE     WED     THU     FRI"
130 FOR O=1 TO 5:PRINTTAB((O-1)*7)M(O);
140 NEXT O
150 END
```

and change line 570 to

```
570 RETURN
```

List the program to confirm that it is now:

```
5 OPEN "RAM:SALEDA.00" FOR INPUT AS 1
10 N = 6
20 FOR W = 1 TO 6 : FOR O = 1 TO 5
30 INPUT # 1, R(W,O)
40 NEXT O : NEXT W
50 FOR O = 1 TO 5 : FOR W = 1 TO 6
60 S(W) = R(W,O)
70 NEXT W
80 GOSUB 500
90 M(O) = MO
100 NEXT O
110 PRINT TAB(10) "MEDIAN SALES"
120 PRINT " MON     TUE     WED     THU     FRI"
130 FOR O=1 TO 5:PRINTTAB((O-1)*7)M(O);
140 NEXT O
150 END
500 FOR I = 1 TO N - 1
510 FOR J = 1 TO N - I
520 IF S(J) < S(J+1) GOTO 540
530 S=S(J) : S(J)=S(J+1) : S(J+1)=S
540 NEXT J : NEXT I
550 N1=INT((N+1)/2) : N2=INT((N+2)/2)
560 MO = (S(N1) + S(N2)) / 2
570 RETURN
```

After the program is entered, execute it. The output should appear as:

MEDIAN SALES				
MON	TUE	WED	THU	FRI
288.5	272.5	325	285	256.5

The median for each day of the week is printed below the name of the day. For each day, the six sales values are sorted and the median computed by averaging the two middle values.

**Line 5** The OPEN statement allows the data to be read from a RAM file.

**Line 10** The numeric variable N is assigned the value 6, indicating six weeks of data for each day of the week.

**Lines 20 - 40** The data is read from RAM and stored in the two dimensional array R. Note that a dimension (DIM) statement was not required in this case since no subscript value will exceed 9.

**Line 50** The first FOR statement in this line begins a loop that calculates the median for each day of the week. Since there are five days, the index variable D runs from 1 to 5.

The second FOR statement in this line begins a loop that transfers the six weeks of data for day into a single dimension array S.

**Line 60** For a given week W and day of the week D, the data value is transferred from the two dimensional array R into the one dimensional array S. The data for the first week will be transferred to S(1), the second week to S(2) and so on.

**Line 70** The inner loop which transfers the six data values from the two dimensional array to the one dimensional array is terminated.

**Line 80** This line contains a new statement: **GOSUB**. When this statement is executed, the program jumps to line 500. Lines 500 through 570 sort and compute the median of the one dimensional arrays S. The median is stored in the variable MD (line 560). When line 570 is executed, the program jumps back to the statement following the GOSUB statement. The block of statements in lines 500 through 570 are called a subroutine. This particular subroutine computes the median of the one dimensional array

S(1), S(2), ... S(N).

The value of the median is stored in the variable MD. It was necessary, in lines 50 - 70, to store column D of the two dimensional array R in the one dimensional array S because the subroutine computes the median of this specific one dimensional array.

**Line 90** The value of the median for day D, which was placed in the variable MD in the subroutine, is stored in the array M(D). This is done because the next median calculated will be placed in MD, replacing the previous value of MD, before the program displays the result.

**Line 100** This line terminates the FOR / NEXT loop which began in the first FOR statement in line 50.

**Line 110** The heading MEDIAN SALES is displayed. The TAB function is used to center the heading.

**Line 120** The names of the five days of the week are displayed in column heading form.

**Lines 130 - 140** The values of the five medians stored in the array M are printed. M(1) contains the median for Monday, M(2) the median for Tuesday, etc.

**Line 150** The **END** statement terminates execution of the program. Without the **END** statement, execution would continue with lines 500 - 570. If that were to happen, the **RETURN** statement in line 570 would generate an error message because there was no corresponding **GOSUB** statement. The **STOP** statement could have been used instead of the **END** statement.

**Lines 500 - 570** This is the subroutine which sorts the array S and computes the median. Before this subroutine can be used (or "called"), the variable S must have the appropriate values assigned to it. Upon returning from the subroutine, the median is stored in the variable MD.

The subroutine can be called as many times as necessary in a program as long as the input variables (N and S) are initialized before it is called. When the **RETURN** statement is executed, control jumps to the statement immediately following the **GOSUB** statement.

Here are some rules governing the use of subroutines.

- 1) *Every subroutine must contain a **RETURN** statement. It may contain more than one **RETURN** statement, if there are several places in the subroutine from which you want to return.*
- 2) *A program may contain several subroutines.*
- 3) *A subroutine may call another subroutine.*
- 4) *A subroutine may be placed anywhere in a program, as long as it is executed only from a **GOSUB** call.*

Subroutines are very useful in programming. They allow you to avoid repetitive blocks of code. They also allow you to write your program in modules or blocks so that it is easier to write and to understand. The program in this experiment was simplified by putting the sorting and median calculation in a subroutine.

## What you have learned:

In this lesson you have seen how arrays, both one and two dimensional, can be used to store data, so that certain information can be extracted. The **TAB** statement was used with the **PRINT** statement to space the output. Two types of averages were calculated, the mean and the median. To calculate the median it was necessary to sort the data. The **INT** function was useful in simplifying the calculation of the median. Finally, subroutines can often be used to make your program easier to write and more readily understandable.



BEEP

SOUND

RND

Animation

---

## Lesson #12 Sound & Simulation

In this lesson you will learn how to create a wide variety of sounds, simulate events using the built-in random number function, and cause apparent movement (animation) on the display.

### Experiment #1 Beep your Beeper!

You can use the built-in speaker to create sounds of many types: beeps, sirens, whistles, clicks, and so on.

Sound can be used in a BASIC program to draw attention to some event — the occurrence of an error, for example. It might also be used to liven-up a program by adding noises to indicate movement. You might also want to use your Computer to create music. All these uses of sound are fairly easy on the Model 100 as you'll find in the following experiments.

You have already seen that printing CHR\$(7) will sound the "bell" character. Try it now to recall the sound:

```
PRINT CHR$(7) (ENTER)
```

Another way to make the same sound is to use the BEEP statement. Type:

```
BEEP (ENTER)
```

and you will hear the same sound. To verify that the sounds are identical, enter

```
PRINT CHR$(7) : BEEP
```

and you will hear two beeps in succession.

You might want to use sound as a sort of warning. Try this:

```
FOR I=1 TO 10 : BEEP : NEXT I (ENTER)
```

You can also vary the tone and duration of the sound using another statement as shown in the next experiment.

### Experiment #2 Sound Off!

Create a tone by entering the following command:

```
SOUND 5586,100
```

You should have heard a tone lasting approximately two seconds and having a frequency of 440 hertz. Increase the frequency of the tone by entering:

```
SOUND 415,100
```

or lower the tone by entering:

```
SOUND 15800,100
```

---

As you can see, the first number in the SOUND statement controls the frequency of the tone and is inversely related to the frequency (the higher the number, the lower the tone).

Increase the duration of the 440 hertz tone by entering:

```
SOUND 5586 , 255
```

and decrease the duration by entering:

```
SOUND 5586 , 1
```

The length of the tone is controlled by the second number in the SOUND statement and ranges from a minimum of 0 (no sound at all) to a maximum of 255. The duration of the tone is approximately 20 milliseconds times the number entered. For example, the command

```
SOUND 5586 , 100
```

turns on a 440 hertz tone for approximately

$$20 * 100 = 2000 \text{ milliseconds} = 2 \text{ seconds}$$

The first number in the SOUND statement determines the frequency of the tone and must be an integer in the range of 0 to 16383. You can hear the full range of tones by entering the following program:

```
10 FOR I = 0 TO 16383 STEP 100
20 SOUND I , 2
30 NEXT I
```

Execute the program.

The sounds you hear are tones of approximately 40 milliseconds duration, ranging from the highest frequency (0) to the lowest frequency (16383) in increments of 100.

The frequency specifier is related to the musical scale, as seen in the table below:

OCTAVE						
Note	1	2	3	4	5	6
C		9394	4697	2348	1171	587
C#		8866	4433	2216	1103	554
D		8368	4184	2092	1045	523
D#	15800	7900	3950	1975	987	493
E	14912	7457	3728	1864	932	466
F	14064	7032	3516	1758	873	439
F#	13284	6642	3321	1660	830	415
G	12538	6269	3134	1567	783	
G#	11836	5918	2954	1479	739	
A	11172	5586	2793	1396	693	
A#	10544	5272	2636	1318	659	
B	9952	4968	2484	1244	622	



You can play the middle C scale by entering the following program:

```
10 FOR I = 1 TO 8
20 READ N : SOUND N, 30
30 NEXT I
40 DATA 4697,4184,3728,3516
50 DATA 3134,2793,2484,2348
```

Execute the program to hear the familiar scale. This program reads eight notes from the data statements and plays them for approximately .6 seconds each. The numbers in the data statements correspond to C, D, E, F, G, A, B and C respectively and were taken from the preceding table.

## Experiment #3 Play a Melody

Referring to the table above, it is rather easy to write a BASIC program to play melodies.

Clear memory using the NEW command and enter the following program:

```
10 READ N,L
20 IF N=0 THEN END
30 SOUND N, L*20
40 GOTO 10
100 DATA 4697,1,4697,1,4697,1,6269,1
110 DATA 5586,1,5586,1,6269,2,3728,1
120 DATA 3728,1,4184,1,4184,1,4697,4
130 DATA 0,0
```

Execute the program and you will hear the familiar Old MacDonald's Farm melody.

**Line 10** The note N and the length L are read from the DATA statements. N is the frequency specifier for the note, and L is the number of beats.

**Line 20** A test for the end of the music is made. If the frequency specifier N is zero, the program ends. Otherwise, the note is played in the next statement.

**Line 30** The SOUND statement plays the note for the desired number of beats. The beat length is approximately

$$20 * 30 = 600 \text{ milliseconds} = .6 \text{ seconds.}$$

**Line 40** Execution jumps back to Line 10 to read the next note.

**Lines 100 - 120** Each note is defined by two numbers: N and L. The frequency specifier N is taken from the table above for the desired musical notes.



The number of beats L is determined by the type of note:

quarter note	= 1 beat
half note	= 2 beats
whole note	= 4 beats

You might like to experiment a little by replacing the DATA statements with your own music. Just remember to terminate the music with a 0,0.

## Experiment #4 Animated Character

Adding sound is not the only way to liven up a program. A display which shows action is often more interesting than a static display. While there are no specific animation statements, you can easily create movement using the PRINT@ statement. The following program illustrates the technique.

Clear memory using the NEW command and enter:

```
10 CLS
20 FOR I=120 TO 158
30 PRINT@ I, CHR$(147);
40 FOR J=1 TO 60 : NEXT J
50 NEXT I
```

Execute this program and watch the stick figure race across the display. While this program added action, it does not adequately simulate movement. You can erase the trail behind the runner by printing a space in his previous position and simulate movement more realistically.

Change line 30 to:

```
30 PRINT@ I, " "; CHR$(147);
```

and execute the program to see a more realistic simulation of movement. This is typical of animation on a computer display; you have to erase the last image and create the new image for each frame in the sequence.

**Line 10** As usual, the program begins by clearing the display.

**Line 20** A FOR/NEXT loop varies the print position from 120 to 158. This corresponds to the fourth line, from the left to the right borders.

**Line 30** Two characters are printed side by side: a space followed by a stick figure. You may wish to confirm that CHR\$(147) is the stick figure by entering

```
PRINT CHR$(147)
```

The space is required to erase the stick figure which was printed on the last cycle of the loop.

**Line 40** A delay loop of a fraction of a second is used to control the speed at which the figure "runs" across the display. You might experiment a little by changing the upper limit in the FOR statement to a smaller value, say 30, and then execute the program. Try the same thing with a larger value, say 200. The upper limit of 60 was determined by trial and error to give a speed which "looked good."

**Line 50** The NEXT I statement determines the end of the FOR / NEXT loop begun in line 20. Note that this program uses nested FOR/NEXT loops.

## Experiment #6 Generating Random Numbers

Computers can be used to simulate random events. This leads to many interesting and useful applications. For example, you can use simulation to create such seemingly diverse applications as interactive games and business decision-making models.

What makes simulation possible on a computer is the ability to generate random numbers. The RND function can be used to return a number between 0 and 1 which can be thought of as "random" in that the numbers appear to occur with equal likelihood and unpredictability.

Clear memory using the NEW command and enter

```
PRINT RND(1)
```

to obtain the random number

```
,59521943994623
```

If you again enter

```
PRINT RND(1)
```

you will get another random number

```
,10658628050158
```

If you continue to print RND(1), you will generate a stream of random numbers.

Enter the program:

```
10 PRINT RND(1) : GOTO 10
```

and execute it.

If you watch the numbers scrolling by on the display, you will notice that they keep changing and that they are all in the range from zero to one.

Press **(BREAK)** to terminate execution of the program. You can change the range of the random numbers quite easily. For example, to generate numbers in the range from zero to 100, simply multiply the RND function by 100. Change the program to

```
10 PRINT 100*RND(1) : GOTO 10
```

and execute it. You should see decimal numbers between zero and 100 scroll by. If you would prefer to have the integers from zero to 100 generated, use the INT function. Change the program to:

```
10 PRINT INT( 100*RND(1) ) : GOTO 10
```

and execute it. You should see integers between zero and 100 scroll by. Using these techniques, you can generate random numbers in any desired range. The next experiment illustrates an application of random numbers.

## Experiment #7 Simulating a Coin Toss

This experiment will use random numbers to create a simulation of tossing a coin. A "head" or "tail" can be generated randomly with the use of RND according to the following scheme:

RND(1)	Outcome
between 0 and .5	Head
between .5 and 1	Tail

Since the probability of generating a number in the range 0 to 0.5 is equal to the probability of generating a number in the range 0.5 to 1, this scheme will generate heads and tails with equal probability.

Use the NEW command to clear memory and enter the following program:

```
10 CLS
20 A = RND(1)
30 IF A < .5 THEN A$="HEAD" ELSE A$="TAIL"
40 PRINT A$,"PRESS ENTER"
50 A$=INKEY$ : IF A$="" THEN 50
60 GOTO 20
```

Execute the program.

If you press **(ENTER)**, you can generate another coin toss. Continue pressing **(ENTER)** a few times to see that the coin tosses give the appearance of a random sequence.

Press **(BREAK)** to terminate execution.

**Line 10** The display is cleared.

**Line 20** A random number is generated and stored in the variable A.

**Line 30** If the value assigned to A is less than .5, then the string "HEAD" is stored in the string variable A\$. Otherwise, the string "TAIL" is stored in A\$.

**Line 40** The outcome is displayed along with a reminder to press **(ENTER)** to continue.

**Line 50** This loop continuously scans the keyboard. When any key is pressed, A\$ will no longer be null and execution resumes on line 60.

**Line 60** Execution jumps back to line 20 to create another coin toss.

Execute and use **(BREAK)** to terminate the program several times. Notice that the program generates exactly the same outcome every time. This is a characteristic of RND(1). Every time a program is run, the same sequence of random numbers will be generated. While this is convenient when you are debugging your program, it does not provide the unpredictability characteristic of true randomness. What is required is a way of starting the sequence of random numbers at an arbitrary point.

Add the following line to the program:

```
15 A=RND(-VAL(RIGHT$(TIME$,2)))
```

List the program to verify that it is:

```
10 CLS
15 A = RND(-VAL(RIGHT$(TIME$,2)))
20 A = RND(1)
30 IF A<.5 THEN A$="HEAD" ELSE A$="TAIL"
40 PRINT A$, "PRESS ENTER"
50 A$ = INKEY$ : IF A$ = "" GOTO 50
60 GOTO 20
```

Execute the program several times to verify that different sequences of outcomes are being generated each time.

Remember to use **(BREAK)** to terminate execution.

The RND function can be used with three types of arguments:

x	RND(x)
greater than 0	generates next random number
equal to 0	generates same random number
less than 0	generates a new sequence of random numbers dependent upon the value of x

In this program, the time is used to determine the sequence of random numbers by negating the seconds of the current time. This will provide 60 different sequences of random numbers. Recall that the time is stored as a string in TIME\$ and must be converted to a numeric value using the VAL function.

## Experiment #8 Slot machine

This experiment will combine sound, animation and random numbers in one program to simulate a slot machine.

Enter the following program:

```
10 A=RND(-VAL(RIGHT$(TIME$,2)))
20 CLS
30 LINE (95,12) - (138,53),1,B
40 LINE (99,22) - (110,33),1,B
50 LINE -(122,22),1,B
60 LINE -(134,33),1,B
```

```

70  FOR T = 1 TO 30
80  SOUND 7000,1
90  A$=CHR$(145+3*RND(1))
100 B$=CHR$(145+3*RND(1))
110 C$=CHR$(145+3*RND(1))
120 PRINT@ 137,A$;
130 PRINT@ 139,B$;
140 PRINT@ 141,C$;
150 NEXT T
160 IF A$=B$ AND B$=C$ GOTO 200
170 PRINT@ 240,"Press ENTER"
180 A$=INKEY$ : IF A$=""GOTO 180
190 PRINT@ 240, "          " : GOTO 20
200 FOR J=1 TO 4
210 PRINT@ 217, "W I N";
220 FOR I=1800 TO 800 STEP -50
230 SOUND I,2 : NEXT I
240 PRINT@ 217, "          ";
250 FOR K=1 TO 100 : NEXT K
260 NEXT J : GOTO 170

```

Execute the program.

A slot machine will appear on the display with three characters appearing at random in the windows. After a few seconds the characters will stop changing. A win occurs if all three characters are the same. The beeping sound is used to help simulate motion in the slot machine. If you don't win, you can press **(ENTER)** to play again. When a win occurs a siren will sound and "W I N" will flash on the slot machine.

**Line 10** The time is used to initialize the sequence of random numbers.

**Line 20** This statement clears the display.

**Line 30** The outline of the slot machine is drawn as a box using the LINE statement.

**Lines 40-60** The three windows are drawn as boxes using the LINE statement.

**Line 70** A FOR / NEXT loop is used to control the duration of the spinning of the slot machine wheels. The upper limit of 30 was determined by trial and error. A larger value would increase the duration and a smaller value decrease it.

**Line 80** A brief tone is generated to suggest rotation of the wheel. Again the parameters were chosen by trial and error to create a reasonable sound.

**Lines 90-110** The characters with ASCII values, 145, 146 and 147 were chosen to be the display characters because they have contiguous ASCII values and they look interesting. These three characters are generated at random with the CHR\$ function

```
CHR$(145 + 3 * RND(1))
```

Since the argument of the CHR\$ is truncated to an integer value, this function will return one of the three characters



---

The string variables A\$, B\$ and C\$ will contain the characters for the first, second and third windows respectively.

**Lines 120-140** The characters are printed in the windows.

**Line 150** The NEXT T statement defines the end of the wheel spinning loop begun in line 70.

**Line 160** If all three characters are the same, execution jumps to line 200, otherwise execution continues with line 170.

**Line 170** The prompt "PRESS ENTER" is displayed in the lower left corner of the LCD.

**Line 180** The keyboard is scanned until a key is pressed.

**Line 190** When a key is pressed, the prompt "PRESS ENTER" is erased and execution transfers to line 20 to repeat the game.

**Line 200** A "W I N" message and a siren are repeated four times.

**Line 210** The word "W I N" is printed on the bottom of the slot machine.

**Lines 220-230** This loop produces the siren sound by increasing the frequency in steps of 50.

**Line 240** The "W I N" message is erased so that it will appear to flash.

**Line 250** A FOR / NEXT loop is used to create a short delay.

**Line 260** The NEXT J statement defines the end of the "W I N" loop. At the end of the loop, execution will transfer to line 170 which will print the prompt and wait for a key to be pressed.

## What you have learned:

You have learned that BASIC programs may be enhanced through the addition of sound, animation and simulation. The SOUND statement allows you to create a tone with a specific pitch and duration. This can be used to generate many different sounds. Motion on the display can be created in a variety of ways, including use of the PRINT@ statement. The RND function lets you simulate events by generating a sequence of random numbers.





## Lesson #13 Function Keys

In this lesson you will learn to use the Function Keys to interrupt execution of a program or to return any desired string of characters.

### Experiment #1 Programming the Function Keys

The keys on the top row of the Model 100 keyboard, labeled F1 through F8, are called **Function Keys**. You have already used some of these in previous lessons. For example, (F1) displays the files saved in RAM, (F4) RUNs a BASIC program, and (F5) LISTs a BASIC program.

These functions were built into the Function Keys to facilitate the use of frequently used operations. However, you can program any Function Key to perform any other operation or to input a string of characters.

Enter the following command:

```
KEYLIST
```

to display the strings programmed into each of the eight function keys. You should see



which indicates that the Function Keys are programmed in the following order:

Key	String	Key	String
(F1)	Files	(F2)	Load "
(F3)	Save "	(F4)	Run
(F5)	List	(F6)	—
(F7)	—	(F8)	Menu

Notice that (F6) and (F7) are not programmed and contain null strings.

Enter the command:

```
KEY 6, "PRINT TIME$"
```

and press the function key (F6). You will see:

```
PRINT TIME$
```

displayed, with the cursor positioned after the word TIME\$. Press (ENTER) to obtain a

KEYLIST

KEY

ON KEY

GOSUB

KEY ON/OFF /  
STOP

display of the current time.

You can eliminate the need to press **(ENTER)** if you also program it as part of the string. You can use `CHR$(13)` to do this.

Enter:

```
KEY 6, "PRINT TIME$"+CHR$(13)
```

and press function **(F6)** to obtain the current time. Notice that it is not necessary to press **(ENTER)** this time since it is programmed as part of the string.

You might wish to program Function Key **(F7)** with the `EDIT` command, since it is so frequently used in BASIC programming.

Enter:

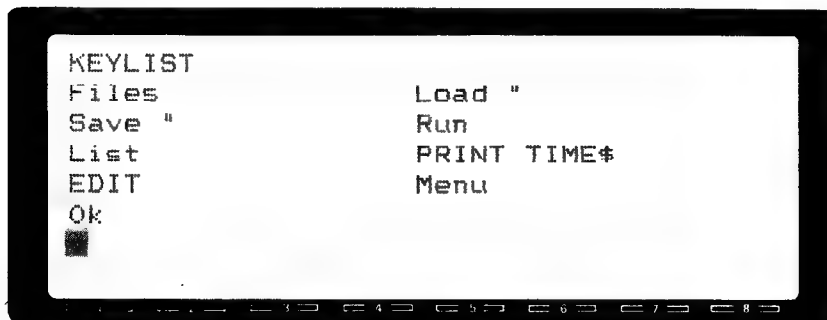
```
KEY 7, "EDIT" + CHR$(13)
```

and press function key **(F7)** whenever you wish to enter the Edit mode. If there is no program currently in working memory, this command will have no effect.

Use the `KEYLIST` command to confirm that you have programmed **(F6)** and **(F7)**. If you enter:

```
KEYLIST
```

you should see



You can change any of the Function Keys, including the factory programmed keys. For example, enter:

```
KEY 3, "PRINT DATE$" + CHR$(13)
```

and press Function Key **(F3)** to obtain a display of the current date. Use the `KEYLIST` command to verify that **(F3)** contains the string

```
PRINT DATE$
```

To program **(F3)** back to the original string

```
Save "
```

you will have to use the `CHR$(34)` form of the quotation marks when you enter the string:

```
KEY 3, "Save " + CHR$(34)
```

Use the KEYLIST command to confirm that you have restored (F3) to its original form. You can restore (F6) and (F7) to their original null string if you wish by entering

```
KEY 6, ""  
KEY 7, ""
```

You can reprogram any function key with any string up to 15 characters. Just remember to use CHR\$(13) for (ENTER) and CHR\$(34) for the quotation mark.

## Experiment #2 Interrupting Execution

You can also use the function keys to control execution of a program. The ON KEY GOSUB statement lets you immediately interrupt a program by pressing a function key. Depending upon the Function Key that is pressed, the program will jump to one of several subroutines.

This interrupt capability allows direct keyboard control of a program during execution. The action is similar to pressing (BREAK), except that execution is not terminated, only redirected.

The program below illustrates how Function Key interrupts work. Clear memory with the NEW command and enter the following program:

```
10 KEY ON  
20 ON KEY GOSUB 100,200,300  
30 PRINT I : I=I+1 : GOTO 30  
100 PRINT "SUBROUTINE 1" : RETURN  
200 PRINT "SUBROUTINE 2" : RETURN  
300 PRINT "SUBROUTINE 3" : RETURN
```

Execute this program and watch the display.

You should see a series of numbers, starting from zero and increasing by one, scrolling on the display.

Press (F1). You should see the message:

```
SUBROUTINE 1
```

displayed and then the numbers will continue scrolling. Press (F2) to display the message:

```
SUBROUTINE 2
```

or press (F3) to display the message:

```
SUBROUTINE 3
```

You can interrupt the printing of numbers at any time using any of the first three Function Keys. Execution jumps to the appropriate subroutine depending upon which Function Key is pressed.

Use (BREAK) to terminate execution of the program.

**Line 10** The KEY ON statement enables the Function Key interrupt capability. In effect, it tells the computer to keep looking for a function key to be pressed. This statement is used in conjunction with the ON KEY GOSUB.

**Line 20** If a function key is pressed, execution will be directed according to the subroutine line numbers in the ON KEY GOSUB statement. For example, if Function Key (F1) is pressed, execution goes to the subroutine in line number 100. If the second Function Key, (F2), is pressed, execution goes to the subroutine in line number 200. If the third Function Key, (F3), is pressed, execution goes to the subroutine in line number 300.

If any other function key is pressed, nothing happens, since there are no more line numbers in the ON KEY GOSUB statement.

The general form of the statement is:

**ON KEY GOSUB** <List of line numbers>

where the list of line numbers corresponds to subroutines. The first line number in the list corresponds to Function Key (F1), the second line number, if present, corresponds to Function Key (F2), and so on.

The ON KEY GOSUB statement must be placed in the program so that it will be executed *before* the Function Key is pressed. Otherwise, the Function Keys will be ignored during program execution.

**Line 30** A continuous loop prints the integers starting at zero and incrementing in steps of one. These statements keep right on printing, so long as a Function Key is not pressed.

**Line 100** This two statement interrupt subroutine is executed if the Function Key (F1) is pressed. The statement which is being executed when the Function Key is pressed is allowed to finish before execution is transferred to the subroutine. This subroutine simply prints the message

SUBROUTINE 1

and returns execution to the statement after the one which was interrupted. For example, if the interrupt occurs while the statement

PRINT I

is being executed, then execution returns to the next statement

I = I + 1.

**Line 200** This interrupt subroutine is executed if Function Key (F2) is pressed.

**Line 300** This interrupt subroutine is executed if Function Key (F3) is pressed.

## Experiment #3 Interrupting an Interrupt

It is possible to interrupt an interrupt subroutine by pressing a Function Key while the interrupt subroutine is executing. You can modify the current program to illustrate this.

Change line 200 to:

200 FOR J = 1 TO 20

and enter the new lines:

```
210 PRINT "SUBROUTINE 2"  
220 NEXT J : RETURN
```

List the program to confirm that it is:

```
10 KEY ON  
20 ON KEY GOSUB 100,200,300  
30 PRINT I : I=I+1 : GOTO 30  
100 PRINT "SUBROUTINE 1" : RETURN  
200 FOR J = 1 TO 20  
210 PRINT "SUBROUTINE 2"  
220 NEXT J : RETURN  
300 PRINT "SUBROUTINE 3" : RETURN
```

Execute this program and press **(F2)** to interrupt execution and begin execution of the interrupt subroutine at line 200.

You should see the message:

```
SUBROUTINE 2
```

repeat 20 times.

When the subroutine is finished printing 20 times, execution returns to the main program. You will see the numbers printing again when this happens.

Again press **(F2)** to interrupt execution of the main program. This time however, press the F1 function key before the 20

```
SUBROUTINE 2
```

messages finish printing. You then should see the message:

```
SUBROUTINE 1
```

displayed once and the SUBROUTINE 2 message continue until all 20 repetitions are complete. Finally, execution returns to the main program where the numbers continue to be displayed.

Sometimes it is desirable to be able to interrupt an interrupt subroutine and sometimes it is not desirable. The **KEY OFF** statement can be used to prevent the interruption of an interrupt subroutine.

Change the following lines in your program:

```
200 KEY OFF:FOR J = 1 TO 20  
220 NEXT J : KEY ON : RETURN
```

List the program to confirm that it is:

```
10 KEY ON  
20 ON KEY GOSUB 100, 200, 300  
30 PRINT I : I=I+1 : GOTO 30  
100 PRINT "SUBROUTINE 1":RETURN  
200 KEY OFF:FOR J = 1 TO 20  
210 PRINT "SUBROUTINE 2"  
220 NEXT J : KEY ON : RETURN  
300 PRINT "SUBROUTINE 3":RETURN
```

Execute the program.

Press the (F2) to interrupt the main program. Again you will see the message:

SUBROUTINE 2

displayed repeatedly on the LCD. While this interrupt subroutine is executing, press (F1). This time you cannot interrupt the interrupt subroutine. However, after the subroutine is finished, the main program can be interrupted with any of the Function Keys (F1), (F2) or (F3).

The reason for this is that the KEY OFF statement in line 200 disabled the Function Key interrupts. The KEY ON, in line 220, restored the Function Key interrupts at the completion of the subroutine.

Rather than totally ignore an interrupt which occurs during an interrupt subroutine, you may wish to have it execute at the completion of the current interrupt subroutine. This can be accomplished with the KEY STOP statement.

Change line 200 to:

200 KEY STOP : FOR J = 1 TO 20

Execute the program.

Press (F2) to interrupt the main program. Then press (F1) to interrupt the interrupt subroutine. Notice that nothing appears to happen. However, if you watch the display carefully, you will see the message:

SUBROUTINE 1

display after the last SUBROUTINE 2 message is displayed.

This is because the KEY STOP statement in line 200 delays execution of the (F1) interrupt until the current interrupt subroutine is finished.

## Experiment #4: Checkwriter with Interrupts

In this experiment you will simulate a payroll program which allows interruption from the keyboard. This could be used to gain access to a data file while the Computer is engaged in a time consuming process. For example, if a payroll program was printing a long list of paychecks, it would tie up the Computer until the printing was completed.

Ordinarily, if you wanted to gain access to the employee data file, you would have to wait until the printing finished or else break the program and then resume printing later. You could use Function Key interrupts, however, to allow immediate inquiry into the file *without* breaking the program. The printing would simply continue after the inquiry was completed.

Clear memory with the NEW command and then enter the following program:

```
10 CLS : KEY ON : ER$=STRING$(30," ")
20 ON KEY GOSUB 140
30 FOR I=1 TO 6:READ N$(I),R(I),H(I):NEXT I
40 FOR I=1 TO 6
50 LINE (12,0)-(226,47),1,BF
```

```

60 PRINT @4, "RADIO SHACK";
70 PRINT @28, DATE$;
80 LINE (24,15)-(210,31),0,BF
90 PRINT @84, "PAY TO: "; N$(I);
100 PRINT @124, R(I) * H(I);
110 PRINT@148,"DOLLARS";
120 FOR K=1 TO 1000 : NEXT K
130 NEXT I : END
140 PRINT @240, ER$;
150 PRINT @280, ER$;
160 PRINT @240, "NAME";
170 INPUT N$
180 FOR J=1 TO 6:IF N$<>N$(J) GOTO 210
190 PRINT@280,"RATE"R(J)"  HOURS"H(J);
200 RETURN
210 NEXT J : RETURN
300 DATA SUE SMITH,7.5,40,TIM LEE,6,35
310 DATA RON REED,8,40,ANN JONES,6,5,38
320 DATA JAN ELY,7,40,SAM BAKER,15,5,38

```

Check the program carefully against the listing and then execute it.

You will see a "check" drawn on the display similar to the one illustrated below:

```

-----
RADIO SHACK                10/10/83
PAY TO:  SUE SMITH
300                                DOLLARS
-----

```

If you let the program run for a few minutes, you will see checks display in succession for each of the employees listed in the data statements. As you can see, it takes quite a bit of time to display each check. This was intended to simulate the time it would take to print a check using a printer instead of the LCD display.

Execute the program again. This time however, press **(F1)** before the second check is displayed. You should see:

```

-----
RADIO SHACK                10/10/83
PAY TO:  SUE SMITH
300                                DOLLARS
-----
NAME?

```

Notice the NAME? prompt just below the check. If you enter the name SAM BAKER, you should see his pay rate and hours worked this week appear on the bottom line of the display as:

```

RATE 15.5  HOURS 38

```

As soon as the rate and hours appear, the checks continue printing. You can interrupt the check printing at any time to find out the pay rate and hours worked of any employee listed in the data statements.

Try interrupting the program and entering several other names. If you should enter a name which is not in the data statements, it will be ignored and printing will continue uninterrupted.

**Line 10** The display is cleared and the Function Key interrupts are turned on. A string of 30 spaces is stored in the string variable ER\$. This will be used to clear a portion of the display.

**Line 20** Function Key (F1) will cause a jump to an interrupt subroutine beginning in line 140. The remaining seven Function Keys remain undefined and will therefore have no effect if they are pressed.

**Line 30** The six names in the data statements are read into the string array N\$(I), the rates are read into the numeric array R(I), and the hours are read into the numeric array H(I). No dimension statement was required in this example since the subscript does not exceed 10.

**Line 40** The FOR/NEXT loop which begins in this line controls the printing of the six checks.

**Line 50** A box is drawn to look something like the outline of a check. The coordinates (12,0) and (226,47) refer to the upper left and lower right hand corners respectively. The ,1,BF specifies a box filled with dark cells.

**Line 60** The company name RADIO SHACK is printed in the upper left corner of the check.

**Line 70** The date is printed in the upper right corner of the check.

**Line 80** An empty box is drawn in the center of the check to make room for the name and amount of the check.

**Line 90** The employee name is printed on the check after the message "PAY TO: ".

**Line 100** The salary is computed as the pay rate R(I) times the hours worked H(I) and printed on the check below the name.

**Line 110** The word "DOLLARS" is printed on the same line as the salary amount.

**Line 120** A delay loop is added to allow time to inspect the check before the next one is printed.

**Line 130** The NEXT I statement is the end of the check printing loop. The program will END after all six checks have been printed.

**Line 140** This is the first line of the interrupt subroutine. Execution will jump here if (F1) is pressed while the main program is executing. The statement in the main program which was interrupted will finish before the interrupt subroutine begins. This statement will erase the first 30 spaces of the next to last line on the display. The purpose of this is to erase any name which may be left here from a previous interrupt.

**Line 150** This statement erases the first 30 spaces of the last line for the same reason given above.

**Line 160** The prompt "NAME" is printed on the next to last line.

**Line 170** This statement allows a name to be entered and assigns it to the string variable N\$.



**Line 180** This FOR / NEXT loop compares the name stored in N\$ to the name stored in the array N\$(I). If the names are different, execution jumps to line 210. Otherwise, line 200 will be executed next. It was necessary to use an index variable other than I in this loop, because I is being used as the index variable in the main program and must not be altered in the interrupt subroutine.

**Line 190** The requested employee's rate and hours are displayed below the name.

**Line 200** This RETURN statement terminates execution of the interrupt subroutine after a rate and hours is displayed. Execution resumes in the main program with the statement following the one which was interrupted.

**Line 210** The NEXT statements define the end of the FOR loop which checks for the name in the DATA statements. If no match is found, this RETURN statement terminates execution of the interrupt subroutine.

**Lines 300-310** These DATA statements contain the six names with their respective pay rate and hours worked.

This simple program suggests how interrupts can be used in a more elaborate payroll program. Typically, a practical payroll program will use a personnel file containing hundreds or perhaps thousands of characters of information for each employee. Conceivably, you could use different Function Keys to inquire about different types of information, for example address, telephone number, number of deductions, length of employment, and so on.

Of course the use of Function Key interrupts is not limited to payroll programs. You may very well find applications in other types of programs including scientific, mathematical, educational and other areas of business.

## **What you have learned:**

You have learned how to program the Function Keys so that a string will be entered with a single keystroke. You have also seen that the Function Keys may be used to control execution of a program through the use of interrupt subroutines.



---

## Lesson #14 Using the COM Option

Your Model 100 has an **RS-232C Interface** which can be used for *serial communications*. In this lesson you will learn how to use the serial port (located in the rear panel of the Computer and labeled RS-232C) to communicate with other devices such as another computer or a serial printer.

Also, many laboratory instruments send results through a serial port making direct data acquisition possible. If the instrument allows two way communication, you might even be able to control the device or the process from the keyboard of the Model 100.

Furthermore, many peripheral devices use serial communications. For example, printers, plotters, voice recognition and synthesis devices, cash registers, modems and EPROM programmers often have an RS-232C Interface for serial communications with a computer. This opens a wide range of possible uses for your Model 100.

To access the serial port, you will need a standard DB25 type connector. Radio Shack offers RS-232C cables in a variety of lengths (such as the five foot cable, catalog number 26-4403). Also, whenever connecting the Model 100 to another TRS-80 Computer, it is necessary to use a **Null Modem Adapter** (26-1496).

### Some Terminology...

To use the serial port (the RS-232C Interface) you don't have to be an expert in serial communications. However, some familiarity with the terminology and concepts would help tremendously. The following discussion highlights some important concepts about serial communications.

Serial communications between a computer and an external device is done one character at a time. A character can be uniquely represented as a series of data "bits," which can be thought of as a list of ones and zeros.

One way of obtaining this bit representation is to express the "ASCII" value of the character in binary form. (ASCII refers to American Standard Code for Information Interchange.) For example, the ASCII value of the uppercase letter "A" is 65 decimal or 01000001 binary. Typically, the number of bits used to represent the ASCII value is 6, 7 or 8.

Since a bit has only two "states," 0 or 1, it can easily be sent over a wire as an electrical pulse where a plus voltage indicates a "1" and a negative voltage a "0." A bit can also be sent as an audio signal, where one tone indicates a "1" and another tone a "0" (as used in a telephone modem).

In **parallel communication**, all the bits that represent the character are sent simultaneously over individual channels. By contrast, in **serial communication** the bits are sent one after the other over the same channel.

In addition to the data bits that represent the character, there are other special bits that are sent over the channel. The data bits are preceded by a *start bit* and may be followed by a *parity bit* and one or two *stop bits*.

OPEN "COM:"

SAVE "COM:"

LOAD "COM:"

ON COM  
GOSUB

COM ON/OFF/  
STOP

INPUT\$

TELCOM

A start bit is always used to signal the beginning of a new character to the receiving device.

A parity bit is sometimes added to provide a means of error detection. If the parity is specified as odd, the parity bit will equal zero if the sum of the data bits is odd, but the parity bit will equal one if the sum of the data bits is even. Thus, with odd parity, the sum of the data bits plus the parity bit is always odd. Similarly, with even parity, the sum of the data plus parity bits is always even.

The stop bit(s) follow the parity bit, if any, and indicate the end of a character to the receiving device. While different systems use various stop bit lengths, the Model 100 allows either one or two stop bits.

Before you can use the RS-232C serial port on the Model 100, you must set the following parameters:

1. **Baud rate, r:** The baud rate is the speed at which the characters are sent. The following values are permitted:

<u>r</u>	<u>Baud rate</u>	<u>r</u>	<u>Baud rate</u>
1	75	6	2400
2	110	7	4800
3	300	8	9600
4	600	9	19200
5	1200		

**Note:** 300 and 1200 baud are common speeds for transmission of data over telephone lines using audio tones. The Model 100 has a built-in modem which allows a direct connection to the telephone lines for 300 baud communications. However, if you use the serial port, rather than audio tones through the modem, you can communicate with other serial devices at any of the baud rates listed above.

2. **Word length, w:** This is the number of data bits used to represent each ASCII character. The three values permitted with the Model 100 are six, seven, or eight data bits.
3. **Parity, p:** The parity bit, if any, is specified as one of:

<u>p</u>	<u>Parity</u>
E	Even
O	Odd
N	None
I	Ignore

4. **Number of stop bits, b:** You can specify either one or two stop bits ( $s = 1$  or  $s = 2$ ).
5. **Line Status (XON/XOFF), s:** Serial communications devices typically use some form of "handshaking" to synchronize transmitting and receiving. This is frequently necessary at the higher baud rates (1200 and above) to prevent transmitting characters to a device which is not ready to receive them.

The Model 100 uses a **handshaking** technique known as **XON/XOFF**. The receiving device sends an XOFF signal to the transmitting device if it cannot

receive any more characters (if a buffer is filled). The receiving device then sends an XON signal when it is ready to receive more characters.

In this way, the receiving device can make the transmitting device wait, as necessary, to give it a chance to catch up. You can enable or disable this feature (the XON/XOFF handshaking) by specifying E, or D respectively.

As an example, the specification

**37N1D**

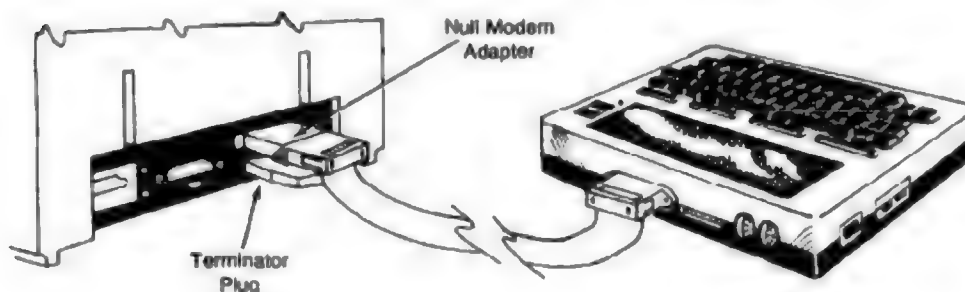
would set the RS-232C serial port for 300 baud, 7 data bits word length, no parity, 1 stop bit and disable XON/XOFF.

## Experiment #1 Transferring a BASIC Program

Suppose you have written a BASIC program in the Model 100 which you plan to use later in your office system.

The following procedure will allow you to transfer a BASIC program to another computer:

1. Connect the two computers via an RS-232C cable (26-4403) and a Null Modem Adapter. See figure below.



2. Set communication parameter in your office system that the Model 100 can match (refer to your system's owner's manual or reference guide).
3. Load the program (in ASCII format) you wish to transfer into the working memory of the Model 100.
4. Set the serial communication parameters to match those of the office system by entering:

**SAVE "COM:RWPBS"**

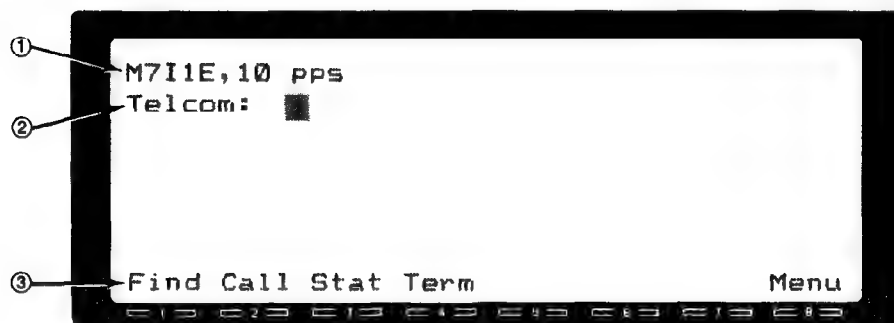
where **R** = baud rate; **W** = word length; **P** = parity; **B** = stop bit; **S** = line status.

**Note:** If a serial printer (rather than another computer) is connected to the serial port, you can use the same command to obtain a listing of the program in the printer.

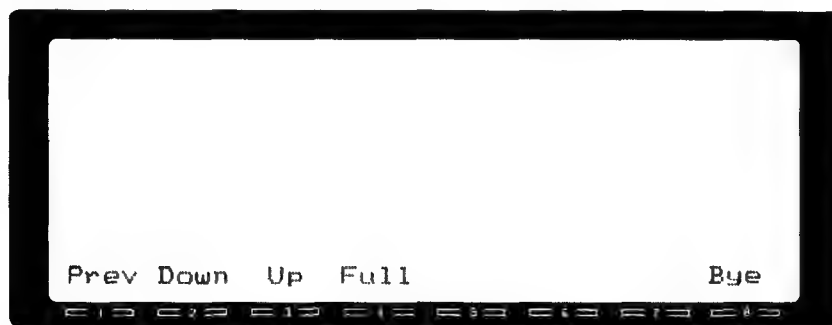
## Experiment #2 Transferring Files to Another Computer using TELCOM

You can transfer any ASCII file to another computer using TELCOM, one of the built-in Application Programs of your Model 100. This includes any BASIC program as long as it has been saved as a RAM file in ASCII format (SAVE "FILENAME",A).

1. Link the two computers together as shown in the previous experiment.
2. Load and execute a communications program that allows data transfer in the host computer.
3. Set communication parameters in the host that the Model 100 can match (refer to your system's owner's manual or reference guide).
4. From the Model 100 Main Menu, position the cursor on the word TELCOM and press **(ENTER)**. The display then shows:



- ① The first line reminds you of the current serial communications status.
  - ② The second line is the TELCOM prompt which lets you select one of the functions displayed on the last line of the screen.
  - ③ The last line shows the definitions of the Function Keys in TELCOM.
5. Change the communication status to match those of the other computer by pressing the Status Function Key (**(F3)**) followed by the desired parameters.
  6. Enter the **Terminal Mode** by pressing the Terminal Function Key (**(F4)**). The bottom line of the display should change to:



The last line indicates the Function Key commands available in the Terminal Mode.

Function Key (F1) now contains a function that allows you to see the "Previous" screen.

Function Key (F2) now contains a feature for **Downloading** or receiving information from another computer.

Function Key (F3) now contains a feature for **Uploading** or sending information to another computer.

Function Key (F4) has now become a **Full/Half duplex** toggle switch. In Full duplex any character that you type on the Model 100's keyboard is first sent to the host computer before it appears on the display. In Half duplex, on the other hand, characters appear on the display just as they are sent to the host.

Function Key (F5), which is not displayed along with the other functions, offers an "Echo" feature which lets you obtain a "hard copy" of whatever is being received (assuming there is a printer connected to the Model 100). Once (F5) has been pressed, it will appear displayed each time you access the Terminal Mode of TELCOM.

Function Key (F6), Bye, lets you exit the Terminal Mode.

7. Now that the computers are connected with matching communication parameters, decide on a file with the extension .DO that you wish to send and press (F3), the Upload Function Key. The prompt:

File to Upload? █

will be displayed.

8. Enter the name of the file to be transferred. The prompt

Width:

will appear. Enter a number between 10 and 132 to format the file as it is sent out. If you simply press (ENTER), the file will be sent "as is."

The word UP on the last line of the display will appear in reverse as the file is being Uploaded. The word will return to normal display when file transfer is complete. If the other computer sends an XOFF command, transmission will pause and the word Wait will appear on the bottom line of the display. If the other computer then sends an XON command to proceed, the word Wait disappears.

**Note:** A BASIC program may be transferred if it has been saved in ASCII, but it cannot be transferred as a .BA file. If you attempt to transfer a .BA file, the error message

NO FILE  
Upload aborted

will be displayed. If you specify a filename which does not exist, the same error message appears.

## Experiment #3 LOADING a BASIC Program

Suppose you have written a BASIC program on another computer which you want to execute on your Model 100. The serial port can be used to transfer the BASIC program to working memory using the following procedure.

1. Connect the computers together as shown in the last two experiments.
2. From the Model 100 Main Menu, enter BASIC.
3. Enter the command

`LOAD "COM:RWPBS"`

where RWPBS correspond to the baud rate, number of data bits, parity, number of stop bits and line status (XON/XOFF) of the other computer.

4. Load and execute a communications program on the other computer which will transfer a BASIC program in ASCII format. This program should send a control Z (ASCII value 26 decimal) as an end-of-file character. If the other computer's program does not terminate the file in this fashion, you can terminate the load manually from the M100 keyboard by pressing **(BREAK)**.

**Note:** **(BREAK)** terminates transfer with an I/O (input/output) error message. Ignore this message. The data has been transferred.

5. Since the program does not display as it is loaded, you will probably want to list it to confirm that it was transferred properly. If a few errors are found, they can be corrected using the Editor. After the file has been successfully transferred, it can be saved in RAM in the usual way.

## Experiment #4 Loading a File with TELCOM

You can transfer any ASCII file to the Model 100 from another computer using TELCOM. This includes a BASIC program so long as it is transferred in ASCII format. The procedure is:

1. Link the Computers through an RS-232C cable and a Null Modem Adapter.
2. From the Model 100 Main Menu, enter TELCOM.
3. If necessary, change the communication parameters to match those of the host system.
4. Press **(F4)** to enter the Terminal Mode.
5. Press **(F2)** to Download a file. The prompt:

`File to Download? █`

will be displayed. Enter a valid filename with either no extension or a .DO extension. If you enter a filename with the .BA extension, the error message:

`Download aborted`

will appear, since BASIC programs can only be transferred in the ASCII format. After the filename is entered, the word Down will appear in reverse video.



6. Load and execute a communications program on the other computer which will transfer the desired ASCII file.
7. To terminate the file transfer and mark the end of the file, press **(F2)** again.
8. To exit the Terminal Mode, press **(F8)** and then **(Y)** when the prompt

Disconnect? █

appears on the screen. If you press **(N)**, you will return to the Terminal Mode.

## Experiment #5 Output to the COM: device

You can use a BASIC program to communicate with peripheral devices through the serial port. If you use the OPEN statement to define a COM: file, you can then direct output to the serial port with a PRINT statement, or input data from the serial port with an INPUT statement. This will be illustrated with examples in the following experiments.

Suppose you want to input a name from the keyboard and then send it out the serial port. From the Model 100 Menu, go to BASIC and enter the following program from the keyboard:

```
10 OPEN "COM:3701D" FOR OUTPUT AS 1
20 INPUT "NAME"; N$
30 PRINT #1, N$
40 GOTO 20
```

**Note:** The peripheral device connected to the RS-232C port should have the same configuration (3701D).

Before executing this program, you should connect a peripheral device with an appropriate RS-232C cable to the M100 serial port and use the correct port status.

Once the peripheral device is connected and ready to receive, execute this program. You will see the prompt:

NAME?

Enter any name, for example

NAME? JONATHAN SMITH

The name will be sent out the serial port and received by the peripheral device. Another prompt will appear on the display to allow transferring as many names as desired. You will have to press **(BREAK)** to terminate this program.

**Line 10** The OPEN statement specifies the COM: device to be used for output as file number 1. The serial port will use 300 baud, 7 data bits, odd parity, 1 stop bit and XON/XOFF disabled.

**Line 20** The INPUT statement prompts for a name from the keyboard and stores it in the string variable N\$.

**Line 30** The name contained in N\$ is sent to the serial port by printing to file number 1.

**Line 40** Execution jumps to line 20 to allow another name to be entered. Lines 20 through 40 form an infinite loop.

## Experiment #6 Input from the COM: *device*

The procedure to input from the serial port is quite similar, as illustrated by changing the previous program as follows:

Enter Edit Mode and change lines 20 and 30 to

```
20 INPUT #1, N$
30 PRINT N$
```

Before executing the program, make sure the peripheral device is properly connected to the serial port and ready to transmit a string of characters. The string should be terminated with a comma or a carriage return and a line feed.

Execute this program, then have the peripheral device send a string. If everything is working properly, you will see the string which was sent from the peripheral device displayed on the Model 100.

Since the program loops back to the INPUT statement, you can have the peripheral device send as many strings as you wish. You will have to press **(BREAK)** to terminate the program.

## Experiment #7 Interrupt from the COM: *device*

You can interrupt execution of a BASIC program from the serial port. This can be very useful to allow communications with a peripheral device which transmits data at unpredictable times. For example, a lab instrument might send test results sporadically as they are completed. Using the interrupt capability, you could use the Model 100 to run a BASIC program, yet still capture the data from the instrument whenever it is sent.

If you have another terminal, such as another computer or a CRT terminal, you might try the following experiment which will serve to illustrate the concept.

Clear memory with the NEW command and enter

```
10 OPEN "COM:3701D" FOR INPUT AS 1
20 ON COM GOSUB 100
30 COM ON
40 PRINT I : I=I+1 : GOTO 40
100 PRINT INPUT$ (1,1) : RETURN
```

Execute this program. You will see numbers starting at zero and incrementing by one displayed along the left margin. These numbers will continue to print until a character is received through the serial port. When this happens, the character received will also be displayed.

If a string of characters is received through the serial port they will all be displayed before the program continues printing the numbers.

**Line 10** The OPEN statement defines the serial port for input as file number 1 at 300 baud, 7 data bits, odd parity, 1 stop bit and XON/XOFF disabled.

**Note:** This configuration should parallel that of the peripheral device. Change the configuration if it is not the same.

**Line 20** An interrupt subroutine beginning at line number 100 will be executed if a character is received through the serial port.

**Line 30** The serial port interrupt capability is turned on. You can prevent the serial port from interrupting the program with the

COM OFF

statement. Similarly, if you wish to delay interrupts, you can use the

COM STOP

statement which defers the interrupt until a COM ON statement is executed.

**Line 40** This loop prints numbers starting at zero and incrementing by one each time. Since it is an infinite loop, you must press **(BREAK)** to terminate the program.

**Line 100** When a character is received through the serial port, an interrupt is generated and execution jumps to this line. No interrupts can occur during an interrupt subroutine. If a character is received during the interrupt subroutine, it will be remembered, and another interrupt will occur when the current one is completed.

This line will display the received character and then return to the main program. The INPUT\$ (1,1) statement inputs one character from file number 1. In general, the function

INPUT\$ (n,f)

will return *n* characters from file number *f*. In the case of the COM: device, the program will wait until all *n* characters are received before it returns.

## Experiment #8 Multiple Character Interrupt

While the procedure in the previous experiment works well with short strings, it will not work if the peripheral device sends strings longer than around 20 characters at a time. You can handle this situation by having the interrupt subroutine input all the characters in the string before returning.

This experiment shows how to modify the program to accommodate longer strings with interrupts. Change line 100 to

```
100 N$ = INPUT$ (1,1) : PRINT N$;
```

and add two new lines

```
110 IF N$<>CHR$(13) GOTO 100
120 PRINT : RETURN
```

List the program to confirm that it is:

```
10 OPEN "COM:3701D" FOR INPUT AS 1
20 ON COM GOSUB 100
30 COM ON
40 PRINT I : I=I+1 : GOTO 40
100 N$ = INPUT$ (1,1) : PRINT N$;
110 IF N$<>CHR$(13) GOTO 100
120 PRINT : RETURN
```

Execute this program and have the peripheral device send a long string of characters terminated with a carriage return. You should see the string correctly displayed on the Model 100.

The program has been modified to keep inputting characters from the serial port while still in the interrupt subroutine. The interrupt subroutine will be terminated in line 120 if a carriage return (ASCII value 13) is detected in line 110. The PRINT statement in line 120 is necessary to print a line feed before returning to the main program.

**A word of caution:** Since the receiving program is written in BASIC, and the serial port buffer has a limited capacity, you still cannot receive long files without missing some characters. This is especially true at higher baud rates.

The procedure described above should therefore be used only with applications requiring relatively short strings to be transmitted at any one time. It is also recommended that you keep the baud rate at 300 or below, if possible.

## What you have learned:

In this lesson you were shown how to use the serial port to communicate with other serial devices. This allows transferring BASIC programs and data files between computers and sending and receiving data between various peripheral devices.

ADDRESS

OPEN "MDM: "

CALL

PEEK

VARPTR

---

## Lesson #15 TELCOM Applications

In this lesson you will learn how to use the **Terminal Mode** of the TELCOM application program and the built-in modem to communicate with other computers over the telephone lines. However, before starting any experiments using the Terminal Mode, it is a good idea to review the overall capabilities of TELCOM.

### TELCOM Overview

As you know from reading the owner's manual, TELCOM allows your Model 100 to be used as an automatic telephone dialer (in the Entry Mode), and to communicate with other computers ('host' systems) over the telephone lines (in the Terminal Mode). The Entry and Terminal modes may also be used together to dial a telephone number and to 'log-on' to a system automatically.

In the Terminal Mode you can access a wide variety of information services, including bulletin boards, news, weather, data banks, other computers or stock market information. You can make use of this information in a variety of ways. For example, you may:

- display it on the LCD
- print it on a printer
- save it in a file
- analyze the data using a BASIC program to graph it, compute statistics, make comparisons against previous values, and more!

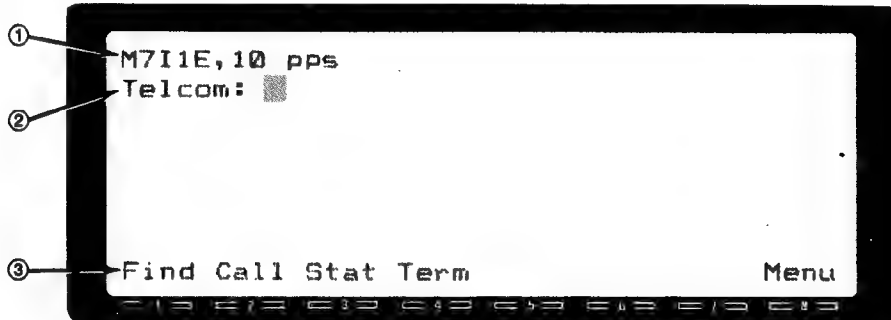
You can also transfer programs or data files to other computers located miles, or even thousands of miles, away.

Before you start using any of the features of TELCOM, the Model 100 must be connected to the telephone lines (consult your owner's manual for detailed instructions). *In this lesson we will assume that your Model 100 is connected to modular telephone lines using the Modem Cable (26-1410).*

*Also, we make the assumption that you have access to a computer bulletin board, or a computer information service (such as Dow Jones or CompuServe). Contact your local Radio Shack Computer Center dealer for assistance in locating a local bulletin board if you cannot locate one on your own.*

## Accessing TELCOM

To access TELCOM, simply move the Cursor over the word TELCOM at the Main Menu and press **(ENTER)**. The display then shows:



When first accessing TELCOM, the Model 100 enters the Entry Mode immediately.

- ① The first line indicates the status of the communication parameters. They appear listed in the order:

- R, baud rate
- W, word length
- P, parity bit
- B, stop bit
- S, line status

The last parameter, 10 pps, indicates the rate for autodialing, ten pulses per second.

- ② The Telcom: prompt in the second line lets you select any of the functions displayed on the last line of the display.
- ③ The last line displays the definition of the Function Keys **(F1)** - **(F8)** in TELCOM's Entry Mode.

For a complete discussion of the role of the Function Keys in the Entry Mode, see "Using the Function Keys in Entry Mode" in the owner's manual (p. 79).

(When entering the Terminal Mode the display of the Function Keys will change to reveal their new uses.)

## Experiment #1 The Communication Parameters

Before entering the Terminal Mode and attempting communications, you must know the communication parameters used by the host system. These are usually provided when you subscribe to an information system and are likely to be listed in your user's guide.

Once you know the communication parameters used by the host system, you must be sure that the Model 100's own parameters match those of the host exactly.

Table 15-1 below, describes the allowable settings for the communication parameters.

Model 100 Communication Parameters		
	You Type:	For:
Baud Rate	M	"modem" (300)*
	1	75 baud
	2	110 baud
	3	300 baud
	4	600 baud
	5	1200 baud
	6	2400 baud
	7	4800 baud
	8	9600 baud
Word Length	9	19200 baud
	6	6 bits
	7	7 bits
Parity	8	8 bits
	I	Ignore parity
	O	Odd parity
	E	Even parity
Stop Bit	N	No parity
	1	1 stop bit
Line Status**	2	2 stop bit
	E	Enable (XON)
Pulse Rate	D	Disable (XOFF)
	10	10pps
	20	20pps

**Table 15-1**

**\* Note:** The Model 100 uses 300 baud when the built-in modem is in use. If you use a number to set the baud rate, even if that number is 3 (for 300 baud), the modem becomes disabled and, instead, the **RS-232C interface** is activated. Therefore, *always* select the letter M whenever the built-in modem is to be used.

If the communication parameters of the Model 100 need to be changed to match those of the host, simply:

1. Access TELCOM.
2. Type STAT, or press **(F3)**, the status Function Key.
3. Type the new communication parameters observing the correct order and press **(ENTER)**.

To verify that the parameters were changed, simply press **(F3)** again and then **(ENTER)**. The new parameters will appear on the display.

---

## Experiment #2 Entering the Terminal Mode

There are two ways to enter the Terminal Mode:

- Automatically via auto-dialing
- Manually via the Term Function Key ((F4))

### Manual Entry

Suppose you wish to contact an information service whose telephone number is 123-4567.

1. Set the **ANS/ORIG** switch (on the left side of the Computer) to **ORIG**.
2. Access **TELCOM**.
3. Lift the receiver and dial the host's telephone number.
4. When the host answers, you will hear a high-pitched tone. Press Term ((F4)).

After pressing Term, the Model 100 will produce a high-pitched tone to indicate it has entered the Terminal Mode.

Also, the definitions of the Function Keys will change as the display shows:



For a detailed discussion of the role of the Function Keys in the Terminal Mode, consult your owner's manual.

### Automatic Entry

To enter the Terminal Mode automatically, you must first store the host's telephone number in the **ADRS.DO** file followed by the special symbols **< >**. This phone number and the symbols **< >** must also be enclosed within colons.

For example, the telephone number of our hypothetical information service would be stored as:

**IS :123-4567 < >:**

where **IS** is your code for the information service.

(If you require extra information for storing telephone numbers in the **ADRS.DO** file consult your owner's manual.)



Then follow these steps:

1. Set the ANS/ORIG switch to ORIG.
2. Access TELCOM.
3. Press Find ((F1)) and type the code for the information service (IS in this case).
4. Press Call ((F2)) after the number appears. You will not have to lift the receiver when calling a host system in this way.

The Model 100 will “echo” a ringing tone or a busy signal (in which case you have to redial the number).

After dialing, the Model 100 produces a high-pitched tone to indicate that it has entered the Terminal Mode and the display changes to reveal the new roles for the Function Keys.

For a detailed discussion of the role of the Function Keys in the Terminal Mode, consult your owner's manual.

## The Log-On Sequence

Once in the Terminal Mode (whether you entered automatically or manually), you must comply with the log-on sequence to gain access to the services offered by the information service.

Most information services will provide you with a **User ID** and a **Password** to serve as confirmation that you are authorized to access a host system.

Detailed instructions for complying with the log-on procedure should be described in your user's guide.

## Experiment #2 Automatic Log-On

The Model 100 gives you the option to combine the Entry and Terminal Modes to dial and log-on to an information service automatically. This is a practical, time-saving feature, especially if you make regular use of an information service.

In this experiment we will create an automatic log-on sequence for the hypothetical information service from the previous experiment. This automatic log-on sequence will serve as a model for creating other log on sequences for real host systems.

Let's assume that IS (Information Service) from the previous experiment specializes in providing information of particular interest to Model 100 owners.

Let's assume that the log-on sequence for gaining access to this system consists of:

1. Sending two carriage returns.
2. Answering the prompt User ID.
3. Answering the prompt Password.

in that order. Your User ID, for this example is 9768,453 and your Password, “Two-tone.” Also, let's assume that after answering these prompts, IS asks you to

select an item of interest, either News or Mailbox. If you so choose, you can also include the selection of the item in the log-on sequence.

Briefly, an automatic log-on sequence consists of identifying the host's log-on prompts and sending the correct responses.

The Model 100 uses a series of **Key Commands** to anticipate and answer the log-on prompts. These commands, listed in Table 15-2 are used as part of any log-on sequence.

TELCOM Auto Log-On Key Commands	
Key	Meaning
?	Wait for a specified character
=	Pause for 2.0 seconds
!	Send a specific character
^	Causes the character after ^ to be sent as a "control" character (i.e., ^M is the same as <b>ENTER</b> )

**Table 15-2**

To log-on automatically, some of the Key Commands, along with the correct responses to the prompts, must be inserted between the < > symbols and stored along with the telephone number in the ADRS.DO file.

After complying with the required responses, the log-on sequence should look like this:

```
IS :123-4567 <^M^M?U9768,453^M?PTwo-tone^M?
      SNews^M>:
```

This is an explanation of how the above log-on sequence was determined:

1. Access the ADRS.DO file and position the cursor over the symbol >.
2. Referring back to the log-on sequence, you must first send two carriage returns. Type ^M ^M (the symbol ^ is obtained by pressing **SHIFT** **6**).
3. Next, tell the Model 100 to anticipate the first prompt from the host, User ID, by typing the Key Command ? (wait for a specific character) and then include a single character from the prompt — we used the letter U.
4. Now, answer the prompt by typing your User ID: **9768,453**.
5. Enter your ID number by typing ^M. You'll recall from the table of Key Commands that ^M is the same as **ENTER**.
6. Tell the Model 100 to anticipate the next prompt, Password, by typing ?P, and then type your response (**Two-tone**). Don't forget to enter this by typing ^M.
7. If you want to include the service you wish to access tell the Computer to wait for the next prompt, What service?, by typing ?W, then follow this with News. Again, you must enter this by typing ^M.

The process for determining other log-on sequences always follows this pattern. For more specific instructions see "Creating an Auto Log-On Sequence" in your owner's manual (p. 91).

After this log-on sequence has been stored in the ADRS.DO file, you can access TELCOM and press FIND (**F1**) to retrieve IS and then call the telephone number by pressing CALL (**F2**). After a few seconds, you'll be logged into the system and have access to the News service.

This is an example of the kind of information you would get:

```
GOOD NEWS FOR M100 OWNERS:
IT WAS LEARNED THAT SEVERAL NEW VENDORS
HAVE RELEASED SOFTWARE PRODUCTS FOR THE
M100 PORTABLE COMPUTER.  CONTACT THEM
DIRECTLY FOR MORE INFORMATION:
```

```
ABC SOFTWARE INC.
1234 ANYSTREET, NEW YORK, NY 00100
```

```
DEF MICROWARE
5678 SCENIC DR, COMPUVILLE, CA 14567
```

```
MACROSOFT
9876 WONDER WAY, WETSVILLE, WA 98734
```

To log-off the system, type

BYE (**ENTER**) or press (**F8**).

A log-off message, such as

```
THANK YOU FOR USING THE M100 USERS
BULLETIN BOARD SYSTEM
DISCONNECTING..
```

may be displayed. Then TELCOM will display the prompt:

```
Disconnect?
```

Enter (**Y**) to disconnect. If you enter (**N**), you will keep the telephone connected and return to the Terminal Mode. If you disconnect the telephone, you will return to the TELCOM system as evidenced by the prompt:

```
Telcom:
```

## Hints and tips...

When you first connect to a computer system, you might notice either of two strange things happen on the display. You might, for instance, see nothing display as you type on the keyboard. This would happen if you have the Model 100 set for Full Duplex and the remote system is set for Half Duplex. If this happens, press (**F4**) to toggle from Full to Half Duplex.

---

On the other hand, you might see every character you type displayed twice. This would happen if you have the Model 100 set for Half Duplex and the remote system is set for Full Duplex. If this happens, press (F4) to toggle from Half to Full Duplex.

You can transfer files over the modem using the Upload and Download capabilities of the terminal. Refer to the previous lesson on serial communications which discusses how to do this and how to use the other features of the terminal mode. The only change required is to specify "M" for the baud rate, R, so that communication is through the built-in modem.

## Experiment #3 Log-On from a BASIC Program

The built-in modem can be used from BASIC to communicate with other devices. The procedure, however, is not especially easy, and is therefore recommended only for advanced or adventuresome programmers. While it is probably easiest to transfer files and communicate with other systems using TELCOM, there are times when it is desirable to use BASIC. For example, you can write a BASIC program to call a stock quotation service on a periodic basis, say every hour, and save the hourly quotes in a data file for later analysis.

The following program segment will dial and log-on to the hypothetical IS (Information Service) from the previous two experiments:

```
100 D$ = "123-4567<^M^M?U9768,453^M?PTwo-tone^M?
    SNews^M>"
110 M = VARPTR(D$)
120 D = PEEK(M+1) + PEEK(M+2)*256
130 CALL 21200 : CALL 21293,0,D
```

**Line 100** The telephone number and-log on sequence are stored in the string variable D\$.

**Line 110** The **VARPTR** function returns an address **M** which helps locate the string D\$. Location **M** contains the length of the string variable, location **M+1** contains the *least significant byte* of the two byte starting address of the string, and location **M+2** contains the corresponding *most significant byte* of the address.

**Line 120** The address of the string variable D\$ is computed and stored in the numeric variable D. The function **PEEK(x)** returns the decimal value of the contents of memory location *x*.

**Line 130** The **CALL** statement is used to call a machine language subroutine. The general form is

**CALL adr,A,HL**

where *adr* is the starting address of the subroutine, **A** is an optional eight bit value to be passed through the A register, and **HL** is an optional 16 bit value to be passed through the HL register. The first statement

**CALL 21200**

calls a machine language subroutine in the Model 100 ROM which "takes the

telephone off the hook." The second statement

```
CALL 21293,0,D
```

calls a machine language subroutine in the Model 100 ROM which dials the telephone number and sends the log-on sequence stored in D\$.

This program is incomplete in that it only dials and logs-on to a host system. There is no provision for further communication with the other system. For example, there is no way to disconnect from the other system or even hang up the telephone.

## Experiment #4 BASIC Communications Through the Modem

The previous experiment showed how to log-on to the hypothetical Information Service, but no provision was made for receiving the NEWS service.

This program segment will display received text after the log-on sequence, send the log-off sequence when an end-of-file character CTRL-Z is received, and hang up the telephone.

```
5 MAXFILES=2
140 OPEN "MDM:7I1D" FOR INPUT AS 1
150 I$=INPUT$(1,1): PRINT I$;
160 IF I$<>CHR$(26) GOTO 150
170 OPEN "MDM:7I1D" FOR OUTPUT AS 2
180 PRINT #2, "BYE" + CHR$(13)
190 CALL 21179
200 CLOSE
```

**Line 5** Since the program has two files open simultaneously, it is necessary to use the MAXFILES statement to provide buffer space for them.

**Line 140** The modem is opened to accept input using file number 1. Note that the status is included in the OPEN statement, but the baud rate is not specified, since it is assumed to be 300 baud through the modem.

**Line 150** Received characters are returned one at a time by the INPUT\$(1,1) function and then stored in the string variable I\$. Each character is displayed by the PRINT I\$; statement.

**Line 160** It is necessary to look for the end-of-file character CTRL-Z to detect the end of the NEWS bulletin. If the character received is not a CTRL-Z ( CHR\$(26) ), execution returns to line 150 to receive the next character. If a CTRL-Z is received, execution resumes with line 170.

**Line 170** The modem is simultaneously opened for output as file number 2 to allow characters to be sent to the bulletin board system.

**Line 180** The log-off sequence BYE plus a carriage return is output to file number 1 (the modem).

**Line 190** A machine language subroutine in the Model 100 ROM is executed to "hang up" the telephone.

**Line 200** Both the input and the output files are closed.

The two program segments can be combined to allow dialing, logging-on, receiving and displaying the NEWS text, logging off and finally, hanging up the telephone. The complete listing would be:

```
5    MAXFILES = 2
100  D$ = "123-4567<^M^M?U9768,453^M?PTwo-tone^M?
      SNews^M>"
110  M = VARPTR(D$)
120  D = PEEK(M+1) + PEEK(M+2)*256
130  CALL 21200 : CALL 21293,0,D
140  OPEN "MDM:7I1D" FOR INPUT AS 1
150  I$=INPUT$(1,1): PRINT I$;
160  IF I$<>CHR$(26) GOTO 150
170  OPEN "MDM:7I1D" FOR OUTPUT AS 2
180  PRINT #2, "BYE" + CHR$(13)
190  CALL 21179
200  CLOSE
```

## What you have learned:

You have learned how to use the built-in modem to communicate with other systems over the telephone lines. You have seen that it is possible to communicate in either TELCOM or BASIC, however, TELCOM is much more straightforward to use. Several machine language-related functions and statements were presented.

# Application #1 Calculator

---

This application program converts your computer into a calculator.

---

Insert the cassette containing the **Calculator** program in your cassette recorder.  
Rewind the cassette if necessary.

Clear memory with the **NEW** command and then load the calculator program by pressing **PLAY** on the recorder and entering the command:

**CLOAD "CALC"**

List the program and compare it to Figure 1-1 to verify that it loaded correctly. Save the program in a RAM file by entering the command:

**SAVE "CALC"**

For convenience, you may also wish to save the calculator program on a separate cassette. Use the command

**CSAVE "CALC"**

to write the program to a blank cassette.

This program allows you to use your Computer as a calculator. This means that numbers may be added, subtracted, multiplied or divided as they are entered from the keyboard, without having to include them as an expression in a **PRINT** statement.

Execute the program. There will be no visual indication that anything has happened. However, if you type the following sequence:

**2+3=**

you will see that the result is immediately displayed:

**2+3= 5**

Notice that this is exactly the way you would enter data to be added on a calculator. Try the following simple calculations:

Press keys in this order	Result
<b>9-2=</b>	<b>7</b>
<b>3.25*7=</b>	<b>22.75</b>
<b>-1/3=</b>	<b>-.3333333333333333</b>
<b>5*6-9.3=</b>	<b>20.7 (30 minus 9.3)</b>
<b>4^2=</b>	<b>16 (4 to the 2nd power)</b>
<b>2+3*4=</b>	<b>20 (5 times 4)</b>
<b>2+(3*4)=</b>	<b>14 (2 plus 12)</b>
<b>(2+3)/(2.5*4)=</b>	<b>.5 (5 divided by 10)</b>
<b>1/((1+ (.5^(-2)))/(1+.5))=</b>	<b>.3 (1 divided by 3.333 . . .)</b>

Notice that exponentiation and parentheses are both supported. Keep in mind that the calculator computes the result immediately as each of the operators is entered. The equal sign (=) prints the result.

---

---

If you make a mistake while entering data in the calculator mode, you can press **(ENTER)** to cancel and start over. You cannot back up to correct an error.

You can use parentheses to make the order of computation unambiguous. The program allows nesting parentheses up to ten deep.

Here is the listing of the Calculator Program:

```
100 R(L)=0 : S$(L)="+"
110 GOSUB 500
120 IF C$="." THEN GOSUB 600 : GOSUB 800
130 IF ASC(C$)>47 AND ASC(C$)<58 THEN GOSUB 600 :
    GOSUB 800
140 IF C$=")" THEN NU=R(L) : L=L-1 :
    GOSUB 800 : GOTO 110
150 IF C$="+" THEN S$(L)=C$ : GOTO 110
160 IF C$="-" THEN S$(L)=C$ : GOTO 110
170 IF C$="*" THEN S$(L)=C$ : GOTO 110
180 IF C$="/" THEN S$(L)=C$ : GOTO 110
185 IF C$="^" THEN S$(L)=C$ : GOTO 110
190 IF C$="(" THEN L=L+1 : R(L)=0 : S$(L)="+" :
    GOTO 110
200 IF C$="=" THEN PRINT R(L) : L=0 : GOTO 100
205 IF ASC(C$)=13 THEN PRINT " CANCELLED" : L=0 :
    GOTO 100
210 GOTO 110
500 REM SUBROUTINE GET CHARACTER
510 C$=INKEY$ : IF C$="" GOTO 510
520 PRINT C$;
530 RETURN
600 REM SUBROUTINE GET NUMBER
630 NU = 0
635 IF C$="." THEN DF=-1 : GOTO 675
640 NU = 10 * NU + VAL(C$)
650 GOSUB 500
655 IF C$="." THEN DF=-1 : GOTO 675
660 IF ASC(C$)<48 OR ASC(C$)>57 THEN RETURN
670 GOTO 635
675 GOSUB 500
677 IF ASC(C$)<48 OR ASC(C$)>57 THEN RETURN
680 NU = NU + VAL(C$) * 10^DF
690 DF = DF-1
720 GOTO 675
800 REM THIS SUBROUTINE COMPUTES THE
810 REM RESULT AT THE CURRENT LEVEL
820 IF S$(L)="+" THEN R(L) = R(L) + NU : RETURN
830 IF S$(L)="-" THEN R(L) = R(L) - NU : RETURN
840 IF S$(L)="*" THEN R(L) = R(L) * NU : RETURN
850 IF S$(L)="/" THEN R(L) = R(L) / NU : RETURN
860 IF S$(L)="^" THEN R(L) = R(L) ^ NU : RETURN
```



## Explanation of the program:

**Line 100** The array R contains the result and the array S\$ contains the most recent operator at level L. The program begins at level 0. Levels are analogous to expressions within parentheses. The level is increased by one as a left parenthesis is encountered, and decreased by one as a right parenthesis is encountered.

**Line 110** A subroutine is used to get a character from the keyboard. The character is returned in the string variable C\$.

**Lines 120 - 130** If the character returned is a decimal point (.) or a digit (0 - 9), then a subroutine (GOSUB 600) is used to build the rest of the number. The value of the number is returned in the numeric variable NU. Another subroutine (GOSUB 800) is used to compute the result of applying the operator to the previous result (if any) and the current value NU. The result is stored in the appropriate level of the array R.

**Line 140** If a right parenthesis is typed, the value at the current level is combined with the value at the previous level. This is why the operator at the previous level was saved in the S\$ array.

**Lines 150 - 185** If an arithmetic operator is typed, it is saved in the S\$ array. Execution then returns to line 110 to get the next character.

**Line 190** If a left parenthesis is typed, the level is increased by one, the result at the new level is initialized to zero, and the operator is assumed to be addition. Execution then returns to line 110 to get the next character.

**Line 200** If an equals sign (=) is typed, then the answer, which is the result at the current level, is displayed. The level is initialized to zero and execution jumps to line 100 to begin a new problem.

**Line 205** If (ENTER) is pressed, then the message CANCELED is displayed and a new problem is begun.

**Line 210** If any other key is inadvertently typed, it must be a mistake, and therefore does nothing to the calculation. Execution jumps back to line 110 to get the next character.

**Lines 500 - 530** A REMark statement is used for documentation in line 500. A REM statement is *ignored* by the BASIC language interpreter and is used to insert explanatory comments for the convenience of the programmer. This subroutine continually scans the keyboard until a key is pressed. The character is stored in the string variable C\$ and displayed before returning. The INKEY\$ function returns a null string

( " " )

if no key is pressed.

**Lines 600 - 720** This subroutine builds a number with or without a decimal point. If the number has a decimal point, the subroutine first builds the part of the number to the left of the decimal point, and then builds the part of the number to the right of the decimal point. Since the characters are originally in string form (C\$), the VAL function is used to convert to numeric form. The number is assumed to be completed when any character other than a digit or a decimal point is typed.

**Lines 800 - 860** This subroutine computes the result at the current level. The appropriate operator (+, -, \*, or ^) is applied and the result stored in the R array.



## Application #2 Memory Master Game

---

This challenging game will test your ability to memorize a random sequence of musical tones.

---

Insert the cassette containing the **Memory Master** application program in your cassette recorder. Rewind the cassette if necessary. If you have loaded the Calculator application program and have not changed the position of the tape, you can speed up loading the Memory Master program by not rewinding the cassette.

Clear memory with the NEW command and then load the Memory Master program by pressing PLAY on the recorder and entering the command

```
CLOAD "MEMO"
```

List the program and compare it to the listing below to verify that it loaded correctly.

```
100 CLS : P=RND(-VAL(RIGHT$(TIME$,2)))
110 PRINT "Want Instructions (Y=yes, N=no)";
    GOSUB 1000
120 IF A$="Y" OR A$="y" THEN GOSUB20000
125 IF A$<>"N" AND A$<>"n" GOTO 100
130 PRINT "Length of sequence L (0-9)"; : GOSUB
    1000 : L = VAL(A$)
132 IF L=0 GOTO 800
134 PRINT "How many notes N (2-9)"; : GOSUB 1000 :
    N = VAL(A$)
136 IF N=0 OR N=1 THEN CLS : GOTO 134
142 NP = 0 : NC = 0 : LAST = SC
145 CLS
150 NP = NP+1
160 FOR I=1 TO L
170 A(I) = INT( N*RND(N) ) + 1
180 FOR J=1 TO 150 : NEXT J
190 SOUND 6000-A(I)*500,20
200 NEXT I
300 PRINT "Enter sequence"
310 C = 0 : FOR I=1 TO L
320 A$=INKEY$ : IF A$="" GOTO 320
322 IF A$<"1" OR A$>"9" GOTO 320
324 SOUND 6000-(ASC(A$)-48)*500,20
330 Y(I) = ASC(A$) - 48
340 IF Y(I)<>A(I) THEN C=1
350 NEXT I
360 IF C=1 GOTO 500
400 PRINT
410 PRINT "C O R R E C T !"
420 NC = NC+1
```

```

430 GOTO 600
500 PRINT
510 PRINT "W R O N G !"
600 PRINT "LISTEN AGAIN" : FOR J=1 TO 500 :
    NEXT J : FOR I=1 TO L
610 SOUND 6000-A(I)*500,20 : PRINTA(I);
620 FOR J=1 TO 150 : NEXT J
630 NEXT I
632 CLS
633 SC = LAST + INT(1000*L*N * NC/NP)
635 PRINT@240,"
640 PRINT@240,"SCORE=";SC
650 PRINT@0,"";
700 PRINT "Want to play again?"
705 PRINT "ENTER=yes, D=change difficulty, S=stop)"
710 GOSUB 1000
712 CLS : PRINT@240,"SCORE=";SC
714 PRINT@0,"";
720 IF ASC(A$)=13 GOTO 150
730 IF A$="D" OR A$="d" GOTO 130
740 IF A$="S" OR A$="s" THEN STOP ELSE GOTO 650
800 REM Press number keys to hear tones
810 CLS
820 PRINT "Press any number key (1 - 9) to"
830 PRINT "hear the tone associated with it."
835 PRINT@280,"Press ESC to begin game";
840 A$=INKEY$ : IF A$="" GOTO 840
850 IF A$<"1" OR A$>"9" GOTO 100
860 SOUND 6000-(ASC(A$)-48)*500,20
870 GOTO 840
1000 A$=INKEY$ : IF A$="" GOTO 1000 ELSE PRINT
    A$ : RETURN
20000 REM ::::: INSTRUCTIONS :::::
20010 CLS : PRINT"MEMORY MASTER is a game which
    tests" : PRINT"your ability to memorize a
    sequence" : PRINT"of tones."
20020 GOSUB 30000
20030 PRINT"Two different skills are being
    tested:" : PRINT : PRINT"1. Your ability to
    recognize different"
20040 PRINT"tones (pitch), and"
20050 PRINT"2. Your ability to recall a random" :
    PRINT"sequence of events." : GOSUB 30000
20060 PRINT"The objective of the game is to
    maximize"; : PRINT"your score, which is
    computed as:"
20070 PRINT"    1000 * N * L * C / T    where"
20080 PRINT"N = number of different tones"
20090 PRINT"L = length of sequence"
20100 PRINT"C = number of correct recalls"
20110 PRINT"T = total number of sequences tried";
    : GOSUB 30000

```

```

20120 PRINT"Beginners should select a sequence" :
      PRINT"length L = 0"
20130 PRINT"which lets you press the number keys" :
      PRINT"1 through 9 to hear the corresponding"
20140 PRINT"tone." : GOSUB 30000
20150 PRINT"When you have learned to associate
      the" : PRINT"number keys with the tones,
      enter a "
20160 PRINT"sequence length L = 1 and number of"
      : PRINT"tones N = 2."
20164 PRINT"Press the '1' key to match the LOW
      tone" : PRINT"or the '2' key to match the
      HIGHER tone."
20166 GOSUB 30000
20170 PRINT"When you have that mastered, increase"
      : PRINT"the length of the sequence L to test"
20180 PRINT"your ability to memorize a sequence,"
20190 PRINT"or increase the number of tones N to"
      : PRINT"test your ability to recognize
      tones : GOSUB 30000
20200 RETURN
30000 PRINT@280,"Press any key to continue";
30010 PRINT@278,""; : GOSUB 1000
30020 CLS : RETURN

```

You may wish to save the program in a RAM file by entering the command

SAVE "MEMO"

For convenience, save the program on a separate cassette with the command

CSAVE "MEMO"

Execute the program.

In answer to the prompt

Want Instructions (Y=yes, N=no)

press ☐ Y to request instructions on the use of the program. The instructions will be presented a few lines at a time. Press any key to continue the instructions. After all the instructions have been displayed you will again see the prompt

Want Instructions (Y=yes, N=no)

Press ☐ N to proceed with the game. You will be prompted to enter the length of sequence L. As suggested in the instructions, press ☐ 0 (zero) to select the practice mode.

As indicated by the display, you may press any of the number keys (1 - 9) to hear the corresponding tone. The larger the number is, the higher the tone will be. The 1 (one) key will give the lowest tone and the 9 (nine) key will give the highest tone. Confirm this by pressing the number keys in sequence (1, 2, ... 9). The game will require you to memorize the tone associated with the keys. You should therefore practice with the keys until you feel comfortable with the association. For beginning levels of play it is only necessary to remember the two lowest tones ☐ 1 and ☐ 2 keys.

Return to the game by pressing **(ESC)**. In fact, if you press any key other than the number keys, you will return to the game as indicated by the request for instructions prompt. When you are ready to play the game, press **(N)** to indicate No instructions.

Enter a **(1)** for the length of sequence L. Enter **(2)** for the number of notes N. The display will clear and you will hear a single tone. Press either **(1)** or **(2)**, depending on which tone you think it is. A message will indicate whether you were correct or not and the tone will be repeated with the correct key displayed. Your score will be displayed.

After each play of the game you have three options:

1. You can play another game at the same level of difficulty by pressing **(ENTER)**. Your score will reflect the percentage of correct responses at this level of difficulty.
2. You can change the level of difficulty and continue to play the game by pressing **(D)**. You can increase the length of the sequence to test your ability to memorize random events. You can increase the number of different tones to test your ability to recognize pitch. Your score from any previous levels of play will be added to the present level score.
3. You can end the game by pressing **(S)**.

### Explanation of the program:

**Line 100** The display is cleared and the random number generator is initialized using the current time in seconds.

**Lines 110 - 125** The user may request instructions or proceed with the game. A subroutine is used to wait for and return a single character from the keyboard. The character is stored in the string variable A\$. A subroutine is used to print out the instructions if requested.

Notice that the IF statements in lines 120 and 125 are testing for both upper and lower case characters. This is a useful technique which you may wish to incorporate in your own programs. Since there is no way to know whether the CAPS LOCK key is depressed or not, this technique ensures a correct response in either case.

Notice also that the IF statement in line 125 traps any key other than Y or N by repeating the prompt. This technique is also useful in general to prevent undesired responses to mis-typed keys. This technique of "fool proofing" should be used whenever possible to prevent undesired program behavior.

**Lines 130 - 132** Another way of trapping inappropriate keyboard input is illustrated. If a non-numeric key is pressed, then VAL(A\$) returns a value of zero. The program will then treat this the same as having pressed **(0)**.

**Lines 134 - 136** The user is prompted to input the number of notes. Another trap is used to prevent an inappropriate response.

**Lines 142 - 200** A sequence of tones of length L is generated. A number from 1 to N is randomly generated and stored in the numeric array A(I). The frequency of the tone, used in the SOUND statement, is related to the random number using the formula  $6000 - A(I)*500$

**Lines 300 - 350** The user is prompted to enter his answer in the form of a sequence of L keys. The tone is heard as each key is pressed (Line 324). A flag is set ( $C = 1$ ) if a key is pressed out of sequence (Line 340).

**Lines 360 - 510** The flag (C) is checked to determine if the user's answer was correct or not. The number of correct responses (NC) is incremented if the answer was correct.

**Lines 600 - 630** The correct sequence of tones is replayed to provide feedback to the user. Note the use of FOR/NEXT loops (Lines 600 and 620) to cause a short pause between tones.

**Lines 632 - 740** The score is computed and displayed. Note that the score is added to the cumulative score (LAST) from any previous level of play. The user is allowed to continue playing at the current level, change the difficulty level or stop the game. Any other response will result in the prompt being redisplayed (Line 740).

**Lines 800 - 870** This section of the program lets the user listen to the tones associated with the number keys (1 through 9). When any key other than a number key is pressed, execution jumps back to the beginning of the program.

**Line 1000** This subroutine continuously scans the keyboard ( $A\$ - INKEY\$$ ) and waits for a key to be pressed ( $A\$$  will be null until a key is pressed). When a key is pressed, it will be stored in the string variable  $A\$$  and displayed before execution returns.

**Lines 20000 - 20200** This subroutine displays the instructions. A subroutine (GOSUB 30000) is repeatedly called to wait for a key to continue the instructions.

**Lines 30000 - 30020** This subroutine displays a prompt to remind you to press any key to continue the instructions. It also calls a subroutine (GOSUB 1000) to wait for any key to be pressed before returning.





## Application #3 Descriptive Statistics

---

This application program computes a variety of common statistics and prints a histogram.

---

Insert the cassette containing the **Descriptive Statistics** application program in your cassette recorder. Rewind the cassette if necessary. If you have loaded the Memory Master application program and have not changed the position of the tape, you can speed up the loading of the descriptive statistics program by not rewinding the cassette.

Clear memory with the NEW command and then load the Descriptive Statistics program by pressing PLAY on the recorder and entering the command:

CLOAD "STAT"

List and compare it to the listing below to verify that it loaded correctly.

```
100 DIM A(100),F(10)
105 CLS
110 INPUT"RAM or CASSETTE FILE (R or C)";T$
120 IF T$<>"R" AND T$<>"C" GOTO 105
130 INPUT"FILE NAME";N$
140 IF T$="R" THEN N$="RAM:" +N$
150 IF T$="C" THEN N$="CAS:" +N$
160 OPEN N$ FOR INPUT AS 1
165 N=1
170 INPUT #1,A(N)
180 IF EOF(1) GOTO 210
190 N=N+1
200 GOTO 170
210 CLOSE
220 CLS
230 INPUT"OUTPUT DATA (Y or N)";T$
240 IF T$="N" GOTO 350
250 IF T$<>"Y" GOTO 220
260 PRINT"OUTPUT ON LCD, LINE PRINTER OR BOTH"
270 INPUT"(L,P or B)";T$
280 IF T$="L" GOTO 310
290 IF T$="P" GOTO 340
300 IF T$<>"B" GOTO 260
310 CLS:FOR I=1 TO N
320 PRINT A(I):NEXT I
330 IF T$="L" GOTO 350
340 FOR I=1 TO N
345 LPRINT A(I):NEXT I:LPRINT " "
350 PRINT:PRINT"SORTING, PLEASE WAIT"
360 FOR I=1 TO N-1
370 FOR J=1 TO N-I
```

```

380 IF A(J)<A(J+1) GOTO 400
390 A=A(J):A(J)=A(J+1):A(J+1)=A
400 NEXT J:NEXT I
410 CLS
420 INPUT"OUTPUT SORTED VALUES (Y or N)";T$
430 IF T$="N" GOTO 550
440 IF T$<>"Y" GOTO 410
450 PRINT "OUTPUT ON LCD, LINE PRINTER OR BOTH"
460 INPUT"(L, P or B)";T$
470 IF T$="L" GOTO 500
480 IF T$="P" GOTO 530
490 IF T$<>"B" GOTO 450
500 CLS:FOR I=1 TO N
510 PRINT A(I):NEXT I
520 IF T$="L" GOTO 550
530 FOR I=1 TO N
540 LPRINT A(I):NEXT I
550 REM CALCULATE THE MEAN
560 FOR I=1 TO N:S=S+A(I):NEXT I
570 M=S/N
580 REM CALCULATE THE (SAMPLE) VARIANCE
590 S=0:FOR I=1 TO N
600 S=S+(A(I)-M)^2:NEXT I
610 V=S/(N-1)
620 REM CALCULATE THE (SAMPLE) STANDARD DEVIATION
630 SD=SQR(V)
640 REM CALCULATE THE MEDIAN
650 N1=INT((N+1)/2):N2=INT((N+2)/2)
660 MD=(A(N1)+A(N2))/2
670 REM DISPLAY THE RESULTS
675 CLS
680 PRINT"NUMBER OF VALUES"TAB(20)N
682 PRINT"MAX, MIN VALUES"TAB(20)A(N);A(1)
684 PRINT"MEAN"TAB(20)M
686 PRINT"MEDIAN"TAB(20)MD
688 PRINT"VARIANCE"TAB(20)V
700 PRINT"STANDARD DEVIATION"TAB(20)SD
702 PRINT:INPUT"OUTPUT RESULTS TO PRINTER (Y or N)";T$
704 IF T$="N" GOTO 740
706 IF T$<>"Y" GOTO 702
708 LPRINT"NUMBER OF VALUES"TAB(40)N
710 LPRINT"MAX, MIN VALUES"TAB(40)A(N);A(1)
720 LPRINT"MEAN"TAB(40)M
725 LPRINT"MEDIAN"TAB(40)MD
730 LPRINT"VARIANCE"TAB(40)V
735 LPRINT"STANDARD DEVIATION"TAB(40)SD
738 LPRINT " ":LPRINT " ":LPRINT " "
740 CLS:INPUT"HISTOGRAM (Y or N)";T$
750 IF T$="N" THEN END
760 IF T$<>"Y" GOTO 740
770 CLS
780 INPUT"NUMBER OF CLASSES (6-10)";NC$

```

```

790 NC=INT(VAL(NC$))
795 IF NC<6 OR NC>15 GOTO 770
800 REM COMPUTE THE FREQUENCIES
810 RG=A(N)-A(1):CL=RG/NC
820 J=1
830 FOR I=1 TO NC-1
840 BD=A(1)+I*CL
850 IF A(J)>BD GOTO 880
860 F(I)=F(I)+1:J=J+1
870 GOTO 850
880 NEXT I
890 F(NC)=N-J+1
900 REM DETERMINE THE MAX FREQUENCY
910 FOR I=1 TO NC
920 IF MF>F(I) GOTO 930 ELSE MF=F(I)
930 NEXT I
940 PRINT"HISTOGRAM ON LCD, LINE PRINTER OR BOTH"
950 INPUT"(L, P or B)";T$
960 IF T$="L" GOTO 1010
970 IF T$="P" GOTO 1110
980 IF T$<>"B" GOTO 940
1000 REM DISPLAY HISTOGRAM ON LCD
1010 CLS
1020 REM DRAW THE AXES
1030 LINE(13,0)-(13,60):LINE-(238,60)
1040 Y2=60:X2=20
1050 FOR I=1 TO NC
1060 X1=X2:Y1=60-50*F(I)/MF
1070 X2=X1+212/NC
1080 LINE(X1,Y1)-(X2,Y2),1,BF
1090 NEXT I
1100 IF T$="L" GOTO 1290
1110 REM PRINT THE HISTOGRAM
1120 X$="*****":Y$=" "
1130 FOR I=30 TO 1 STEP -1
1140 LPRINT "*";TAB(10);
1150 FOR J=1 TO NC
1160 H=30*F(J)/MF
1170 IF H>I THEN LPRINT X$; ELSE LPRINT Y$;
1180 NEXT J:LPRINT
1190 NEXT I
1200 LPRINT STRING$(80,"*")
1210 FOR I=1 TO NC
1220 LPRINT TAB(10+4*(I-1))I;:NEXT I
1230 LPRINTY$:LPRINTY$:LPRINTY$
1240 LPRINT"CLASS NUMBER"TAB(29)"CLASS"TAB(49)"
FREQUENCY
1250 LPRINT STRING$(60,"-")
1260 FOR I=1 TO NC
1265 BD=A(1)+(I-1)*CL
1270 LPRINT TAB(4)I;TAB(14)BD;TAB(31)"TO";BD+CL;
TAB(51)F(I)
1280 NEXT I
1290 END

```

---

Save the program in a RAM file by entering the command:

```
SAVE "STAT"
```

You may wish to save the descriptive statistics program on a separate cassette to make loading faster in the future. Use the command

```
CSAVE "STAT"
```

to write the program to a blank cassette.

This program computes a variety of common statistical measures on a set of data. The data must be in the form of a file either in RAM or on cassette. You can use TEXT to create the data file.

Go to the Menu by pressing (F8). Move the cursor over the word TEXT and press (ENTER). You will see the prompt

```
File to edit?
```

Enter a name for the data file to be created. For example, enter a file name of DATA:

```
File to edit? DATA
```

Type a list of data values separated with either commas or carriage returns. You can speed up this process by pressing (NUM) and then using the appropriate keys as a ten-key number pad.

For example, type:

```
23,5,35,7,45,2,46,12,67,30,44,22,33,45<
16,37,56,74,82,13,28,74,66,91,43,32,55<
46,77,31,86,58,59,62,43,71,38,69,23,77<
48,39,42,36,45,30,37,43,47,41<
```

Save the data as a RAM file by pressing (F8). You will return to the Menu. If you want to make any changes to your data file, simply move the cursor over the file name DATA.DO and press (ENTER). Use the standard Editor commands to make your changes.

After your data file is created, you can use the descriptive statistics program to list it, sort it, compute the number of values, the maximum and minimum, the mean, median, variance and standard deviation. You can also plot a histogram of the data to give an indication of its distribution.

From the Menu, place the cursor over the file name STAT.BA and press (ENTER). This will load and execute the descriptive statistics program. You should see the prompt:

```
RAM or CASSETTE FILE (R or C)?
```

If you have saved your data in the RAM file DATA.DO as described above, enter an R. You will see:

```
RAM or CASSETTE FILE (R or C)? R
FILE NAME?
```

Enter the name of your data file. For the example, enter a file name of DATA. The display will clear and another prompt is displayed:

```
OUTPUT DATA (Y or N)?
```

If you would like the data file to be listed, enter ☐Y (for Yes). Enter ☐N (for No) if you do not want to see the data listed.

If you do request the data to be output, you will see the prompt

OUTPUT ON LCD, LINE PRINTER, OR BOTH  
(L, P or B)?

Here you have a choice of output devices. Enter L if you want the data displayed only on the LCD, enter P if you want the data output to the printer only, or enter B if you want the output to go to both the LCD and the printer.

If you request output to the printer, make sure that you have your printer attached to the printer port and that it is turned on. Otherwise, the program will wait for the printer before proceeding.

The data is displayed and/or printed in a single column as illustrated below:

23.5  
35.7  
45.2  
...  
...  
43  
47  
41

After the data is displayed, or if no data display was requested, you will see the message

SORTING, PLEASE WAIT

After the data has been sorted, the display will clear and you will be prompted with:

OUTPUT SORTED VALUES (Y or N)?

Enter ☐Y (for Yes) if you want to output the data values in sorted order, or enter ☐N (for No) if you do not desire a listing of the sorted data values.

If you do request the sorted values to be output, you will again be prompted to direct the output to the LCD, the Printer or both as shown below:

OUTPUT SORTED VALUES (Y or N)? Y  
OUTPUT ON LCD, LINE PRINTER, OR BOTH  
(L, P or B)?

The data values will be output in ascending order as illustrated below:

12  
13  
16  
...  
...  
82  
86  
91

After the sorted data has been output, or if no output was requested, the display will clear and the results will be displayed. For the example, you would see:

NUMBER OF VALUES	47
MAX, MIN VALUES	91 12
MEAN	47.2
MEDIAN	44
VARIANCE	387.63895852174
STANDARD DEVIATION	19.68849807684

The bottom line of the display gives the prompt

OUTPUT RESULTS TO PRINTER (Y or N)?

If you would like to output the above results to the line printer, enter ☐ (for Yes). Enter ☐ (for No) if you do not want the results printed.

After the results are printed, or if you do not request output to the printer, the display clears and another prompt appears:

HISTOGRAM (Y or N)?

Enter ☐ (for Yes) if you would like to see a histogram (graphic plot of the distribution of the data). Enter ☐ (for No) if you do not want a histogram.

If you request a histogram, you will be prompted to enter the number of classes:

NUMBER OF CLASSES (6-10)?

The number of classes is the number of intervals of equal width to classify the data into. For example, if you want to divide the data into seven equal intervals, enter 10.

You must now direct the histogram to be output to the LCD, the Printer or Both in response to the next prompt:

NUMBER OF CLASSES (6-10)? 10  
HISTOGRAM ON LCD, LINE PRINTER OR BOTH  
(L, P or B)?

If you enter ☐ (B), you should see the histogram on the LCD similar to Figure 3A-1 and on the printer similar to Figure 3A-2.

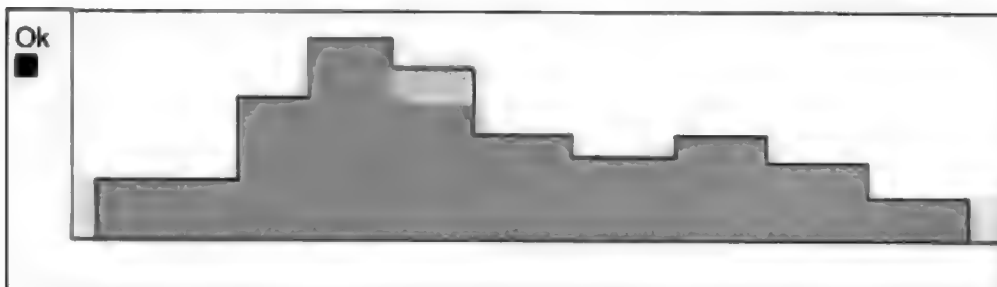
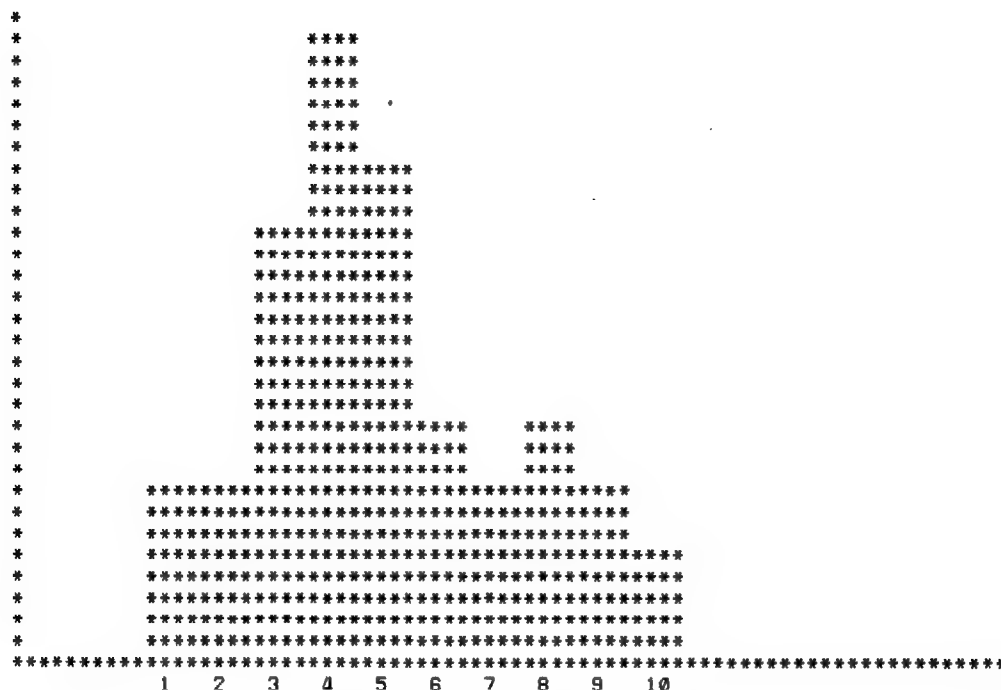


Figure 3A-1. LCD Histogram



If you request printer output, you will also obtain a listing of the class intervals and the corresponding frequencies of occurrence within each interval. This is illustrated in Figure 3A-3. The frequency is the number of data values which occur within the class boundaries. For example, in Figure 3A-3, there are seven data values within the approximate boundaries 27.8 to 35.7.

CLASS NUMBER	CLASS	FREQUENCY
1	12	3
2	19.9	3
3	27.8	7
4	35.7	10
5	43.6	8
6	51.5	4
7	59.4	3
8	67.8	4
9	75.2	3
10	88.1	2

### Explanation of the Program:

**Line 100** The data values will be stored in the array A and the frequencies for the histogram will be stored in the array F. The maximum number of data values is 100.

**Lines 105 - 150** This part of the program allows the user to indicate whether the data will be read from a RAM or cassette file and to input the file name. The file name is stored in N\$ along with the device (either RAM: or CAS:).

**Lines 160 - 210** The data file is opened for input. The data is read one value at a time from the input file until an end of file is encountered (**Line 180**). When the last data value is read, the input file is closed. The number of data values is incremented and stored in the numeric variable N.

**Lines 220 - 345** The user is asked whether to output the data values or not. If not, execution jumps to Line 350. If output is desired, the user is asked to specify whether the LCD, the Printer or Both should be used. If the LCD only is desired, only the PRINT statement is executed. If the Printer only is desired, only the LPRINT statement is executed. If both the LCD and the Printer are desired, then both the PRINT and the LPRINT statements are executed.

A FOR/NEXT loop increments through all N data values.

**Lines 350 - 400** A bubble sort is used to put the data values into ascending order. This is necessary in order to compute the median.

**Lines 410 - 540** This section is similar to Lines 220 - 340, except that the sorted data values will be output if desired.

**Lines 550 - 660** The mean, variance, standard deviation and median are calculated and stored respectively in the variables M, V, SD, MD.

**Lines 670 - 738** The results are output to the display using PRINT statements. The user is asked if output is also desired to the printer. If it is, the results are output a second time using LPRINT statements.

**Lines 740 - 930** The user is asked if a histogram is desired. If not, execution ends. If a histogram is desired, it is necessary to input the number of classes, NC, and then to calculate the class length, CL. A FOR/NEXT loop (Lines 830 - 880) increments through all data values and counts the frequency of occurrence in each class. The frequencies are stored in the F array.

A FOR/NEXT loop (Lines 910 - 930) is used to compute the maximum frequency and stores it in MF. The maximum frequency is required in order to scale the histogram display.

**Lines 940 - 980** The user is asked whether to output the histogram to the LCD, to the Printer or to both.

**Lines 1000 - 1100** This section of the program displays the histogram on the LCD. Since the display area is limited, no labels are used on the axes. The LINE statement is used to draw the axes.

A FOR/NEXT loop (Lines 1050 - 1090) is used to draw a box for each class, where the height of the box represents the frequency of that class.

**Lines 1110 - 1220** The histogram is output to the printer. Since there is no LINE statement for use with the printer, the histogram must be printed one line at a time using standard ASCII characters. The asterisk, "\*", was picked somewhat arbitrarily as the fill character because of its relative density.

A FOR/NEXT loop (Lines 1130 - 1190) prints the "boxes". The vertical axis is printed in line 1140. The horizontal axis is printed in line 1200. A FOR/NEXT loop (Lines 1210 - 1220) is used to print the horizontal axis labels.

**Lines 1230 - 1280** A table consisting of the class intervals and the corresponding frequencies is printed.



# INDEX

Subject	Page	Subject	Page
ADDRESS .....	182	Entry Mode .....	175
AND .....	120, 121	Error Message .....	3
ANS/ORIG .....	178	FILES .....	16
ASCII .....	26, 116-118, 165	FOR/NEXT .....	88, 90, 91
ATN .....	114	FRE .....	115
Adding (a line) .....	12	File Extension .....	16
Arithmetic Expressions .....	32	File Name .....	15, 16
Arithmetic Operators .....	32	Files .....	3
Arrays .....	92-94	Full/Half Duplex .....	169, 181
Assignment Statement .....	32	Function Keys .....	155
Auto Log-On .....	179	GOSUB .....	142, 143
BASIC .....	1, 3	GOTO .....	6
BASIC Program .....	5	Handshaking .....	166
BEEP .....	145	IF/THEN .....	46
BREAK .....	3, 6	IF/THEN/ELSE .....	50-52
Baud Rate .....	166	INKEY\$ .....	119
Bits .....	165	INPUT .....	35-38
Branching .....	41	INPUT\$ .....	173
Bubble Sort .....	136	INPUT# .....	125
CALL .....	182	INT .....	135, 140
CHR\$ .....	116	Infinite Loop .....	6, 54
CLEAR .....	40, 41	Insert (a line) .....	12
CLOAD .....	22	Insert Mode .....	71
CLOAD? .....	24	KEY ON .....	155
CLOSE .....	124, 125	KEY OFF .....	159
CLS .....	88	KEY STOP .....	160
COM ON .....	172, 173	KEY .....	155, 156
COM OFF .....	173	KEYLIST .....	155, 156
CONT .....	7	KILL .....	28, 29
CSAVE .....	21	Key Commands .....	180
Cassette .....	20	LEFT\$ .....	58
Command .....	4	LEN .....	60
Communication Parameters .....	176	LINE .....	100
Condition .....	47	LIST .....	7, 16
Cursor Movement Keys .....	70	LOAD .....	17
Cut .....	82, 83	LOAD "CAS: filename" .....	23
DATA .....	55, 56	LOAD "COM: .....	170
DATE\$ .....	65	Line Status .....	166
DAY\$ .....	57	Log-ON Sequence .....	179, 181
DIM .....	92, 93	Log-Off .....	181
Delete (a line) .....	11	Log-On .....	175, 182
Deleting .....	74	Logical Operators .....	121
Download .....	169, 182	Looping .....	53
EDIT .....	70	MAXFILES .....	128
END .....	143	MERGE .....	26-28
ENTER .....	3, 4	MID\$ .....	63
Echo .....	169	Main Menu .....	3
Editing .....	7	Multiple Statements .....	90
Editor .....	69	NAME .. AS .....	20
end-of-file .....	70	NEW .....	13

Subject	Page
NOT .....	125
Nested Loops .....	97
Numeric Constants .....	32
Numeric Variables .....	32
ON COM .....	172
ON KEY GOSUB .....	157, 158
OPEN .....	124
OPEN "COM: .....	171
OPEN "MDM: .....	183
OR .....	121
PAUSE .....	8
PEEK .....	182
PRESET .....	99
PRINT .....	4
PRINT USING .....	48
PRINT @ .....	101
PRINT # .....	124
PSET .....	99
Parallel Communication .....	165
Parity bit .....	165, 166
Password .....	179
Paste .....	83, 84
Paste buffer .....	82
RAM .....	15
REM .....	187
RESTORE .....	55
RIGHT\$ .....	64
RND .....	149
RS-232C Interface .....	165
RUN .....	5, 6
Random Numbers .....	149
Relational Operator .....	47
Reverse Video .....	82
SAVE .....	16, 17
SAVE "CAS:filename" .....	21
SAVE "COM: .....	167
SOUND .....	145, 146
SOUND OFF .....	145
SQR .....	111
STEP .....	96
STOP .....	56
STR\$ .....	63
STRING\$ .....	61
SUBROUTINE .....	157
Select .....	82, 83
Serial Port .....	165
Serial Communications .....	165
Sorting .....	136, 137
Start Bit .....	165
Stop Bit .....	165-167

Subject	Page
Subscripted Variables .....	87, 93
Syntax Error .....	4
TAB .....	131
TAN .....	113
TELCOM .....	175
TIME\$ .....	62
Term .....	178
Terminal Mode .....	168, 175, 178
Uploading .....	169, 182
User ID .....	179
VAL .....	63
VARPTR .....	182
Variable, String .....	40
Word Length .....	166
XON/XOFF .....	166, 167



